NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

VISUAL NAVIGATION
FOR AN
AUTONOMOUS MOBILE VEHICLE

by

Kevin Robert Peterson

26 March 1992

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited

VISUAL NAVIGATION
FOR AN
AUTONOMOUS MOBILE VEHICLE

Kev... ...erson
Lt., ...
B.S. Optics, Univers... ...ster, 1986

Submitted in partial fulf...
requirements for the degr...

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

Author: ______________________
Kevin Robert Peterson

Approved By: ______________________
Yutaka Kanayama, Thesis Advisor

____
Neil C. Rowe, Second Reader

____
Robert B. McGhee, Chairman,
Department of Computer Science

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
VISUAL NAVIGATION FOR AN AUTONOMOUS MOBILE VEHICLE (U)

12 PERSONAL AUTHOR(S)
PETERSON, KEVIN ROBERT

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 03/90 TO 03/92	1992, March 26	

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Robotics, image-understanding, pattern-matching, pose-determination

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Image understanding for a mobile robotic vehicle is an important and complex task for ensuring safe navigation and extended autonomous operations. The goal of this work is to implement a working vision-based navigation control mechanism within a known environment onboard the autonomous mobile vehicle *Yamabico-11*. Although installing a working hardware system was not accomplished, the image processing, model description, pattern matching, and positional correction methods have all been implemented and tested on a graphics workstation. A novel approach for straight-edge feature extraction based upon least squares fitting of edge-related pixels is presented and implemented for the image processing task. A simple method for determining the camera's location and orientation (*pose*) follows by matching the vertical extracted edges from an image with the linear features of a two-dimensional view of the modelled environment based upon an estimated pose of the robot. Image processing, construction of the two-dimensional view of the model, and pose dtermination are conducted sequentially in less than one minute for a 646 x 486 pixel image on a 35 MHz processor. The pose determination results have been tested to be accurate within a few inches for tranlational error and within one degree rotational error.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Yutaka Kanayama	22b. TELEPHONE (Include Area Code) (408) 646-2095 / 22c. OFFICE SYMBOL CS/Ka

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ABSTRACT

Image understanding for a mobile robotic vehicle is an important and complex task for ensuring safe navigation and extended autonomous operations. The goal of this work is to implement a working vision-based navigation control mechanism within a known environment onboard the autonomous mobile vehicle *Yamabico-11*. Although installing a working hardware system was not accomplished, the image processing, model description, pattern matching, and positional correction methods have all been implemented and tested on a graphics workstation. A novel approach for straight-edge feature extraction based upon least squares fitting of edge-related pixels is presented and implemented for the image processing task. A simple method for determining the camera's location and orientation (*pose*) follows by matching the vertical extracted edges from an image with the linear features of a two-dimensional view of the modelled environment based upon an estimated pose of the robot. Image processing, construction of the two-dimensional view of the model, and pose determination are conducted sequentially in less than one minute for a 646 x 486 pixel image on a 35 MHz processor. The pose determination results have been tested to be accurate within a few inches for translational error and within one degree rotational error.

TABLE OF CONTENTS

I. INTRODUCTION

A. MOTIVATION FOR VISION DEVELOPMENT

Image understanding has been a difficult subject in the fields of artificial intelligence and robotics for nearly twenty years. The prospect of developing a novel and revolutionary approach to image processing and image understanding in the course of writing a master's thesis seems unlikely. Instead, as with most all vision systems developed, this approach will be directed towards developing a visual navigation system based upon the specific needs of the robot and tailoring the application to the domain of its operating environment. Certain assumptions will be made to simplify the problem. These assumptions are:

- the robot will operate in an indoor, orthogonal environment,
- the floor will be a flat surface,
- the floorplan of the environment will be known,
- the robot will always be in an upright position,
- and approximate position information will be accessible.

The image understanding techniques to be described have been developed utilizing a charge-coupled device (CCD) television camera, a graphics workstation, and a video frame grabber. The CCD television camera (JVC model TK870U) is capable of providing digitized RGB formatted color information (red, green, and blue components) to an image processing system. The graphics workstation is a SiliconGraphics Personal Iris and the video frame grabbing board is SiliconGraphics' VideoFramer. The SiliconGraphics graphics library supports basic display functions in the C language. The vision based routines described within this thesis are programmed in the ANSI C language.

B. AUTONOMOUS MOBILE ROBOT SYSTEM YAMABICO-11

Yamabico-11 (Figure 1.1) is an autonomous mobile robot that has been the focus of many students' theses at the Naval Postgraduate School. Work in the fields of motion control, safe path planning, and sonar-based navigation are continually ongoing to upgrade the robot's capabilities. Yamabico's platform is an aluminum cart that stands 36 inches tall, with two main wheels for motion and steering and four smaller wheels for support. Its maneuvering capabilities resemble that of a tracked vehicle (it can pivot in place). Power is provided from two 12-volt wheelchair batteries.

Process control is conducted by one Motorola 68020 32-bit microprocessor with an accompanying Motorola 68881 floating point coprocessor. The processor has 1Mbyte of onboard RAM and runs at 16MHz. Control information is passed over a VME bus which also carries a dual axis controller board and two 4-

channel serial communication boards. The dual axis controller interfaces with driving and braking motors. The serial channels interface with an onboard terminal and a modem to a host computer.

Its only sensors, twelve ultrasonic sonars that operate at 40kHz, are mounted about the lower base and are controlled by three Motorola 6809-based microprocessors. The Sonar Control Language [SHE91] and the Mobile Motion Language (MML) [KAN89a] provide easy sonar and motion control features to the programmer and are downloaded to Yamabico via an RS232 serial port from the host computer, a Sun Microsystems 3/160. The system is composed of a *kernel* and a *user-program*. The kernel (about 100 kbytes) only needs to be downloaded once. The user-program, *user()*, specifies Yamabico's tasks. Once the object code has been downloaded and the serial link disconnected, all computation is performed autonomously and is executed via a keyboard mounted on top of Yamabico.

Presently, Yamabico can navigate its way through a "known" corridor by utilizing ultra-sonic range information to update its position which is maintained by wheel motion dead-reckoning. Yamabico's ultra-sonic sensors are effective in ranges from 9.6cm to 400cm and are accurate to 18.95mm at a range of 500mm. The range information can directly update the current position in a known environment (such as our testing grounds, the fifth floor of Spanagel Hall at the Naval Postgraduate School). Prior to the work outlined in this thesis, Yamabico had no image based sensor capabilities at all. The purpose of this thesis is to develop and implement a vision based sensor system for Yamabico-11 suitable for navigation and exploration of its environment.

Figure 1.1 The autonomous mobile robot *Yamabico-11*.

C. LITERATURE SURVEY

Image understanding has been a complex topic for computer scientists for almost two decades. The ideas and algorithms utilized in this work stem from the image processing, linear fitting, pattern matching, and pose determination methods that have been found to be successful in past applications. Two good sources for the basic ideas necessary in image understanding are by Ballard [BAL82] and Nevatia [NEV82]. These are both textbooks that outline the more commonly used methods and the background fundamentals. They cover a wide range of topics including low level image processing, graph-theory for pattern matching, and model-base object recognition.

A paper by Marr [MAR79] outlines requirements for model-based image understanding and provides insight for construction of suitable models, similar to his "2D+," model for vision applications. Stein's thesis [STE92] describes the environment model developed for the vision navigation method developed in this thesis. Two papers by Kanayama, [KAN90a] and [KAN90b], were consulted for familiarization with the robot Yamabico's motion control capabilities (with the MML language) and the application of linear fitting of sonar point data by least squares for environment feature mapping. Sherfey's thesis [SHE91] provides the details on Yamabico's sonar capabilities.

A lot of recently published research conducted at the University of Massachusetts at Amherst provided motivation to develop faster and simpler image edge extraction and visual navigation routines. Papers from this group by Beveridge [BEV90], Kumar [KUM89], and Fennema [FEN90] describe methods for pattern matching, pose determination, and their experimental results. Edge extraction was conducted via the Hough transform and pattern matching performed by random combinations of image to model line matches.

Pose determination methods described by Haralick [HAR89] and Liu [LIU90] are algorithms based upon the general trigonometric principles of photogrammetry. A photogrammetry textbook, such as Moffit's [MOF80], proved to be a good source to explain these fundamentals. During development of this algorithm, a means for pose verification was found to be necessary. Methods tested by Heller [HEL89] show the results of verifying all image lines with model lines and the results of verifying model lines with edge-related pixel data. A basic computer graphics textbook by Hall [HAL89] was also consulted for checking video source formats and computer image data formats.

D. THESIS ORGANIZATION

The following Chapters outline the development of the algorithm applied towards a simple visual navigation scheme for an autonomous mobile robot in a known orthogonal environment. The objectives are stated in Chapter II. Chapter III presents the general method of feature extraction based upon determining the edges depicted in an image. Chapter IV discusses the method in which the image's pixels are grouped into two-dimensional regions describing the edge [illegible] of straight line segments from two-dimensional areas based upon least-squares fitting of datapoints is [illegible] Chapter V. The straight edge extraction algorithm is then outlined in Chapter VI. The algorithm's re[illegible] [illegible]fects of varying parameters within the algorithm are shown in Chapter VII. Chapter VIII discusses imp[illegible] factors such as image data structures and the environment model.

Chapter IX discusses the method for pattern matching of extracted edges from [illegible] with modelled environment features. Chapter X describes *pose* (the vehicle's position and orientation within the environment) determination based upon disparities between the image and the expected view of the modelled environment from an assumed position. Pose determination results are shown in Chapter XI. Conclusions and subsequent topics for this visual navigation system are presented in Chapter XII.

Appendix A specifies the atan2 function used for determination of a vector orientation and it contains the data types header files used. Appendix B contains the edge extraction implementations *findedge* and *fastedge* and all the required header files for compilation. Appendix C contains the implementation *vertmatch* which extracts vertical image lines, matches them with the model lines from an expected two-dimensional view of the environment, and determines the robot's correct position and orientation. Appendix D contains a user's guide for the compilation and operation of the edge extraction and pose determination programs.

II. OBJECTIVES

There are two primary objectives for the development of a vision understanding system for the Yamabico-11 robot. They are similar to the objectives for any autonomous mobile vehicle.

1. Update correct vehicle position and orientation for precise navigation.
2. Recognize specific objects using *a prior* knowledge based upon the task requirements and the operational environment.

A. POSITION / ORIENTATION IDENTIFICATION IN A KNOWN WORLD

Determining accurate position and orientation autonomously is a crucial requirement for the successful and safe navigation of any mobile robot. This task is also known as *pose determination* and is used to update the robot's positional information that is maintained by wheel motion dead-reckoning. Small errors from dead-reckoning arise quickly and are compounded by turns and acceleration. Continuous periodic pose determination is fundamental for precise dead-reckoning based navigation. Chapter X outlines a simple position-correction algorithm via pattern-matching of the extracted features from a two-dimensional video image and two-dimensional view of mapped features of the three-dimensional environment model maintained by the robot. This algorithm is based upon previous works [HAR89], [MAR79], [MOF80], [KUM89], [FEN90], and [LIU90], whose roots stem from shipboard navigation and aerial photogrammetry.

B. OBJECT RECOGNITION IN AN UNKNOWN OR PARTIALLY KNOWN WORLD

The second objective is the recognition of specific objects via three-dimensional models. This is a much more complex task than the first due to the uncertainty of individual objects presence and location. The first task of pose determination is a subset of this task. The operating environment is modelled as a three-dimensional object and determining the camera's pose is only determination of viewing aspect of the object. Object recognition covers many other topics including mapping an unknown environment, searching for a specific goal, classifying encountered objects, and avoiding unknown obstacles. Some aspects of object recognition are detailed in [BAL82], [MAR79], [NEV82], and many others. A central theme of all vision-based object recognition applications is modelling and searching for only the objects that are expected to be encountered and classifying unknown items by their geometrical properties. This objective is not approached within this paper, but will hopefully be accomplished for Yamabico by a successor.

III. METHOD

A. ARCHITECTURE FOR IMAGE BASED NAVIGATION

Extracting desired features from an image is a crucial process in any image understanding implementation. Some methods focus on determining areas of common light intensity, color, or texture. These methods often include segmentation of the image by region growing and are generally suitable for images with small areas of pixels with common attributes (e.g. a picture of a forest). Other methods search for the edges between contrasting bordering areas of pixels with similar attributes. Likewise, these methods are suitable for images with large areas of common light intensity, color, or texture (e.g. a picture that included bare walls). This method will be of the latter category; extracting the edge features of an image. This method was chosen for recognizing the straight edges of orthogonal objects whose surfaces are often uniform in color and texture. The images that the edge extraction methods will be applied to, will be pictures of the interiors of offices and hallways. Pictures where straight line segments would be common and suitable for describing the objects in the image.

The goal of the image processing part of the vision system will be to find straight line segments in the edges of boundaries between areas of similar light intensity in an image. The straight line segments should be a sufficiently simple data structure for use in follow-on image understanding implementations. The specific problem of image understanding for navigation of Yamabico in a known, orthogonal environment will be split into the following subproblems:

1. Image Processing - extracting desired features (straight line edges) from the input image.
2. Pattern Matching - correlating extracted features with known features described in a three-dimensional model of the environment.
3. Pose Determination - calculating the position and orientation of the robot within its modelled operating environment.

Since Yamabico has no vision based sensor capabilities, the following features were identified for the implementation of a vision understanding system (Figure 3.1)

- a three-dimensional model representation of the operating environment,
- camera image processing functions,
- image feature extracting functions,
- pattern matching routines,
- and visual-based position correction methods.

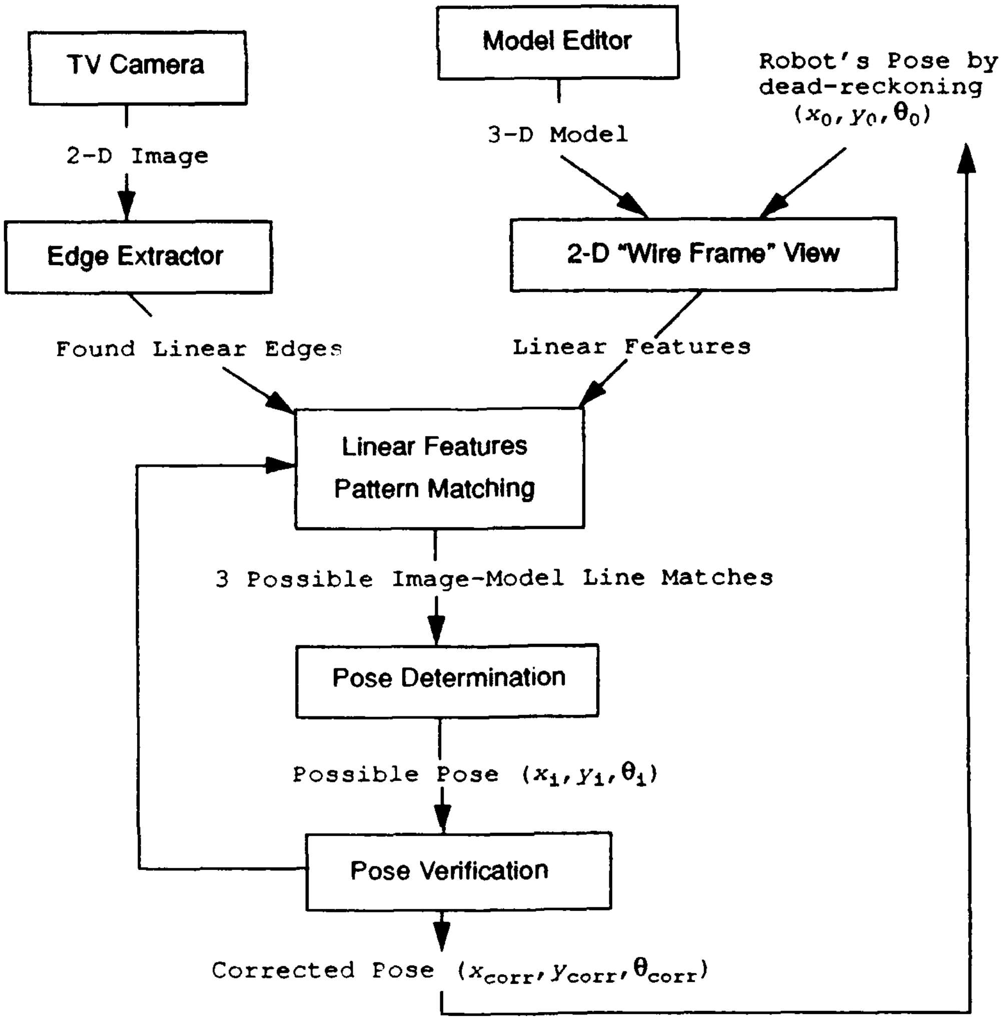

Figure 3.1. The general outline for image understanding on Yamabico-11.

In order to accomplish the image processing task of finding straight line segments in an image, the borders (edges) between areas will be identified, segmented, and then simplified in terms of straight line segments (Figure 3.2). The identification of the edges in an image (Figure 3.3) is best described by a *gradient image* (Figure 3.4). Two important steps to follow are:

1. grouping of pixels into contiguous regions of separate edges, and
2. linear fitting of the edges.

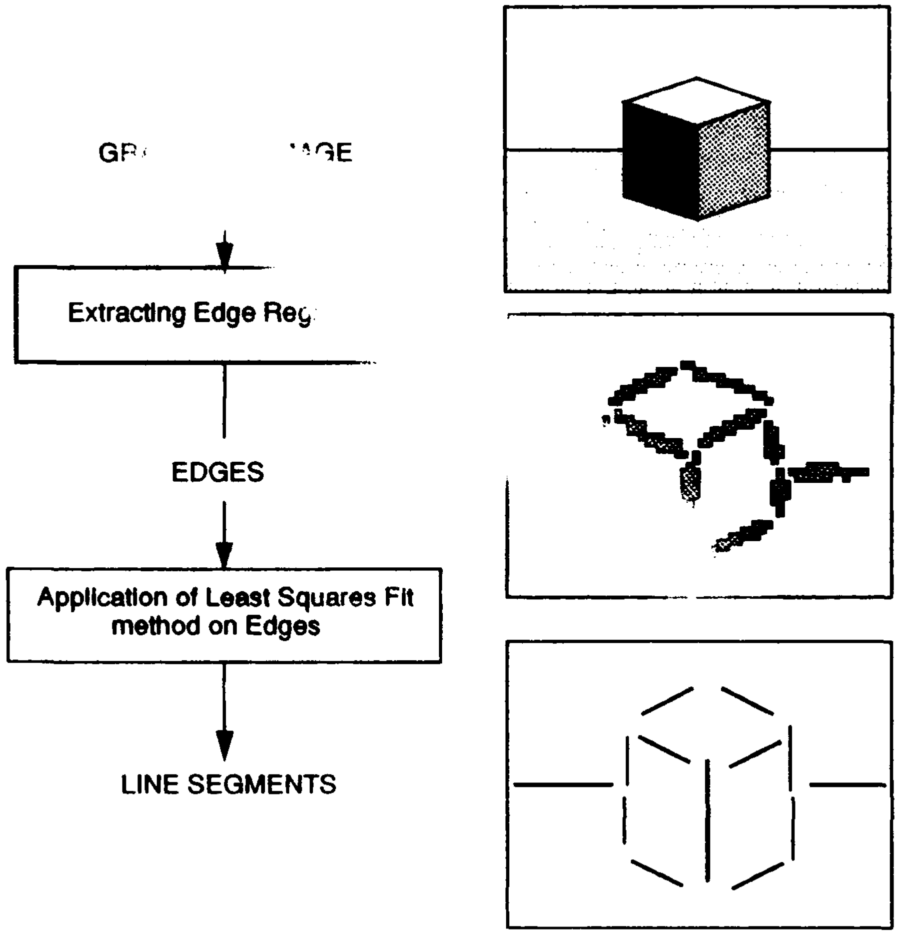

Figure 3.2. Outline of method for edge extraction of an image. Edges will be represented as line segments.

Thus, once the gradient image is made, pixels that define the edges (the black pixels of Figure 3.4) must be grouped and then fitted to a straight line segment by least squares fit to the major axis of the edge region. Determining when to start and stop constructing separate edges is a crucial subject. Checking the validity of the line segments will be the final requirement of the image processing task.

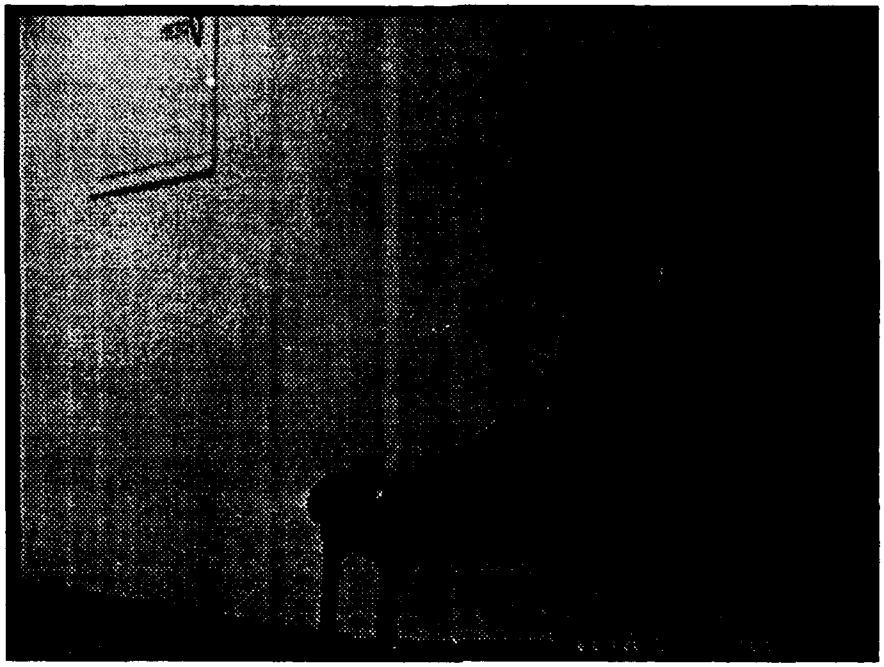

Figure 3.3. An input image of a chair.

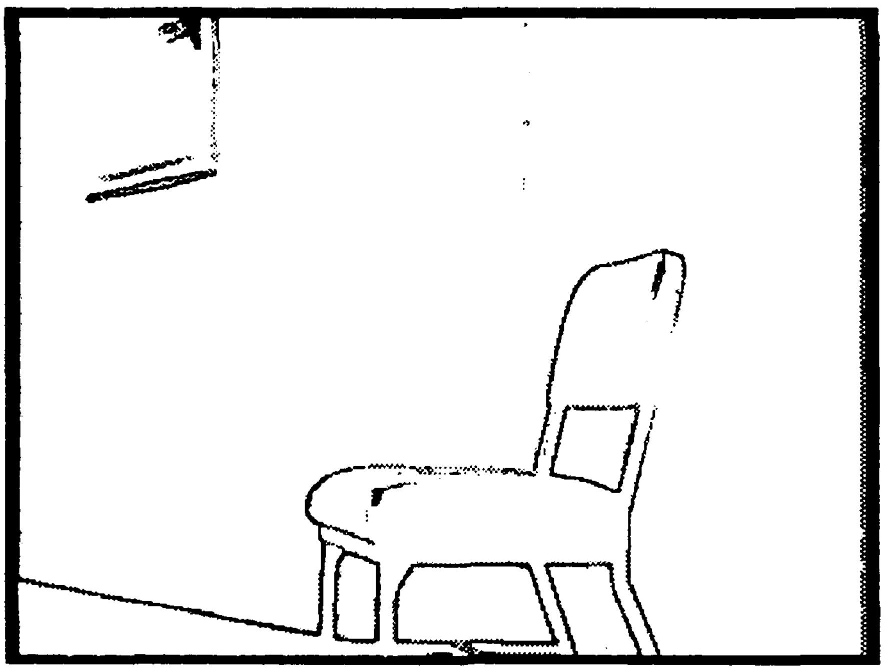

Figure 3.4 The gradient image for the chair in Figure 3.3. Edges between regions of common light intensities are depicted by black pixels.

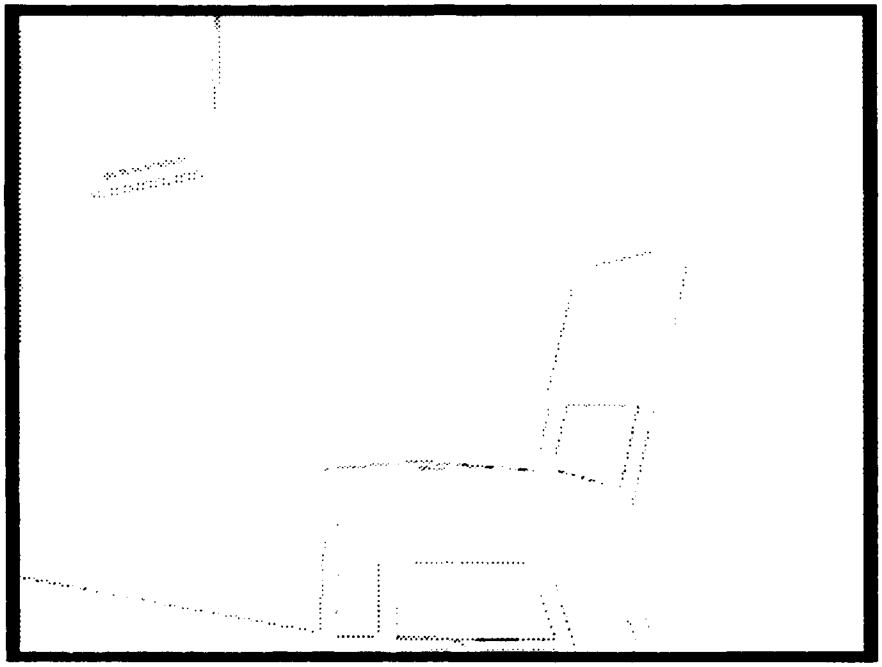

Figure 3.5. The extracted line segments based upon the grouping of similar edge-related pixels.

The pattern matching process will require two inputs. These will be the linear features from the three-dimensional model mapped onto a two-dimensional viewing plane and edges extracted from the image processing system. The three-dimensional model is a two-dimensional floorplan plus vertical components for height measurement. This model will be used to construct a two-dimensional "wire-frame" view of the environment from any given pose for the robot's camera. This view will be used for what the robot will expect to see from its present location and orientation. The three-dimensional modelling system for Yamabico is being pursued concurrently with this work by Lt. Jim Stein for his master's thesis [STE92].

The image processing system could be executed in parallel with the three-dimensional modelling system as it provides an updated two-dimensional wire-frame view of the robot's orthogonal environment. The pattern matching process will correlate the extracted edges from the image with the known features from the model.

The task of pose determination will follow the matching of the three most significant edges with all possible vertical model lines. The observed difference of horizontal viewing angles between the vertical image edges will be fitted with the map locations of vertical model lines to determine the only possible position and orientation of the robot's camera. Many different possible values for the robot's pose will be calculated as different line matchings are tested.

A method for pose verification will be necessary to select the best possible pose (hence, also the best matching) to be returned as the correct pose of the robot. The correct pose can then be used to update the pose maintained by dead-reckoning. Sensor fusion with the present sonar system could possibly follow for confirming the accuracy of the visual fix for the pose.

IV. GENERATING REGIONS OF EDGES

A. GRADIENT IMAGE

The first requirement for the image edge extract is the determination of high contrast regions between the areas of similar light intensity within regions of significant light intensity contrast will define the edges. The CCD camera simplifie ion of light intensity by discretization of the image into separate pixels. Chapter VII identifies som nts provided by the camera and used in the graphics workstation. The pixel format utilized herein rmat which specifies the red, green, and blue color attributes of the pixel and is described fully in Chap ferences between the areas of the image are therefore determined by differences in pixel intensities. The simplest way to consider pixel intensities is in terms of *grayscale* values vice considering the separate associated color attributes of each pixel. The color to grayscale conversion of RGB pixels in also described in chapter VII. Determining the differences in grayscale intensities between adjacent pixels in a two-dimensional image will result in a gradient value for each pixel. The pixel gradient is a vector whose magnitude represents the amount of light intensity change between adjacent pixels, and whose angular orientation is directed towards the lighter pixel. Pixel gradient values will be determined in a two-dimensional cartesian coordinate plane by the amount of change of grayscale intensities in both the horizontal (dx) and vertical (dy) directions of the image.

1. Gradient Window Operators

A common method to determine pixel gradients is by the use of gradient window operators. Two gradient window operators, one for determining the change of pixel intensities in the horizontal (dx) direction and another for the vertical (dy) direction, must be specified. Gradient window operators are square matrices of weights, mapped onto a group of pixels about a central pixel ghts are multiplied with the intensities of the surrounding pixels and then e intensity changes in the horizontal and vertical axes. Commonly use uare, Prewitt, Sobel, and Schirai (Figures 4.1 through 4.4).

-1	1
-1	1

dx

1	1
-1	-1

dy

Figure 4.1. The Four-square gradient windows.

-1	0	1
-1	0	1
-1	0	1

dx

1	1	1
0	0	0
-1	-1	-1

dy

Figure 4.2. The Prewitt gradient windows.

-1	0	1
-2	0	2
-1	0	1

dx

1	2	1
0	0	0
-1	-2	-1

dy

Figure 4.3. The Sobel gradient windows.

0	1	1
-1	0	1
-1	-1	0

dx

1	1	0
1	0	-1
0	-1	-1

dy

Figure 4.4. The Schirai gradient windows.

To be geometrically correct for a two-dimensional plane of pixels evenly spaced in both horizontal and vertical directions, a modified Sobel operator should be used with values of $\sqrt{2}$ vice 2 for weights of the non-diagonal pixels (Figure 4.5). This would assume that all of the pixels sensors on the CCD array in the camera are in a perfectly square grid. However, since one of the assumptions of this vision system is to operate in an orthogonal environment, the standard Sobel operator could prove to be more valuable since it enhances the vertical and horizontal edges found in the orthogonal environment. After implementing and testing all of these window operators, the Sobel seemed to provide the strongest edges and the least amount of smaller, insignificant and undesirable edges.

-1	0	1
$-\sqrt{2}$	0	$\sqrt{2}$
-1	0	1

dx

1	$\sqrt{2}$	1
0	0	0
-1	$-\sqrt{2}$	-1

dy

Figure 4.5. The modified Sobel gradient windows.

Once the values for dx and dy are calculated by the gradient window operators, the gradient is easily calculated. Pixel gradient magnitude ($|\nabla\ (pixel_i)|$) and gradient angle ($\nabla\ \alpha(pixel_i)$) would then be determined by equations 4.1 and 4.2. The *atan2(dy,dx)* function is common in many programming languages' standard math libraries and is described in Appendix A.

$$|\nabla\ (pixel_i)| \equiv \sqrt{(dx_i)^2 + (dy_i)^2} \tag{4.1}$$

$$\nabla \alpha(pixel_i) \equiv \text{atan2}\,(dy_i, dx_i) \tag{4.2}$$

To determine if a pixel is part of an edge between two areas of an image, the gradient magnitude must be greater than a specified threshold value, C_l. Such a threshold can be determined dynamically by scanning through the image once to determine the average weight of pixel intensities or by maintaining a histogram of previous images' average pixel intensities. An alternative approach is to keep the gradient magnitude threshold value as static value, determined by testing. This method would be suitable for a robot operating in an environment with relatively constant illumination and alleviates the requirement of an extra scan through the image to dynamically determine a value for C_l. This method will be pursued based upon the assumption that the robot will initially be tested in an indoor environment with relatively uniform lighting, whereas the other method for threshold calculation would be more suitable for the vision system adjusting to major environment lighting changes (e.g. the robot travelling into an unlit room).

2. Construction of the Gradient Image with the Sobel Operator

The Sobel gradient window operators will be utilized to calculate pixel gradients from the grayscale values for each pixel. The Sobel operator is defined as a two-dimensional matrix; that is to be implemented upon pixels read and stored in a one-dimensional array (see Chapter VIII for description of the data structure for the image). Using the image's horizontal dimension (*xsize*) as an offset for finding the adjacent pixels in the neighboring rows of a two-dimensional image, equations for calculating the dx and dy values for the gradient of a central pixel, i, are defined by equations 4.3 and 4.4. A pointer (*ptr*), is utilized to access each grayscale pixel value from the one-dimensional array in the data structure for the image.

$$dx = \begin{array}{c} -ptr\,[i + xsize - 1]\ + ptr\,[i + xsize + 1] \\ -2ptr\,[i - 1]\ + 2ptr\,[i + 1] \\ -ptr\,[i - xsize - 1]\ + ptr\,[i - xsize + 1] \end{array} \tag{4.3}$$

$$dy = \begin{array}{l} ptr\,[i+xsize-1]\ +2ptr\,[i+xsize]\ +ptr\,[i+xsize+1] \\ -ptr\,[i-xsize-1]\ -2ptr\,[i-xsize]\ -ptr\,[i-xsize+1] \end{array} \qquad (4.4)$$

Notice that in order to calculate the Sobel operator for pixel *i*, pixel *i+xsize+1* and all previous pixels must have a grayscale values already determined. A possible method could be to first construct a corresponding grayscale image of the input (color) image followed by construction of the gradient image using the Sobel operator equations. Pixel gradient magnitudes larger than the predefined threshold (C_1) would then be set to be black in the gradient image and [illegible] would be white. This would provide a gradient image such as displayed in Figure 3.4. A simple alg[illegible] this would be:

- Read in input image.
- Allocate memory for new grayscale and gradient
- Build the grayscale image by grayscale conversion [illegible]
- Build the gradient image via Sobel operator for all pixels.
- If pixel gradient magnitude is greater than the magnitude threshold, C_1, then set the corresponding pixel in the gradient image black, else set the corresponding pixel white.

3. When Not to Use the Sobel Operator

However, not every pixel in the input image (more precisely, the grayscale image) can have their associated gradient values properly calculated. The outermost pixels, those in the top and bottom rows and the leftmost and rightmost columns will not have gradient values since the Sobel operator requires eight adjacent pixels about the central pixel gradient value. Calculating gradient for these outermost pixels will only produce errors when using the gradients. Therefore [illegible] ent image are two pixels less in both horizontal and vertical directions and [illegible] be paid towards scanning through the input image in order not to calculate pixel gradients for the outermost pixels.

B. FINDING CONTIGUOUS REGIONS OF PIXELS TO DESCRIBE EDGES

Since the black pixels in the gradient image represent pixels with gradient magnitudes sufficiently large to be considered part of an edge, each black pixel must be considered for grouping into contiguous regions that define the edges. Therefore, grouping pixels into regions will be performed upon every pixel with a gradient magnitude larger than the threshold, C_1. Determining a suitable value for C_1 is critical to the success of this algorithm. A threshold that is too high will not allow for the detection of all significant edges with those that are found being broken and incomplete. On the other hand, a threshold that is too low lead to

edge finding and linear fitting of non-edge areas of the image. Chapter VII shows results of varying the gradient magnitude threshold C_1.

After a suitable threshold is determined and used, the problem of grouping edge-related pixels is to define how two pixels share features common to an edge. Figure 4.6 can be considered as a portion of a gradient image with two distinct areas of common light intensity (areas A and B) and the subsequent edge in between, comprised of pixels with gradient magnitudes greater than the given threshold. In order to describe the edge, each pixel must be grouped into one contiguous region. There are three important aspects of a pixel that can be used for this comparison: pixel gradient magnitude, pixel gradient angle, and pixel adjacency. Pixel gradient magnitude has already been considered by testing with threshold C_1.

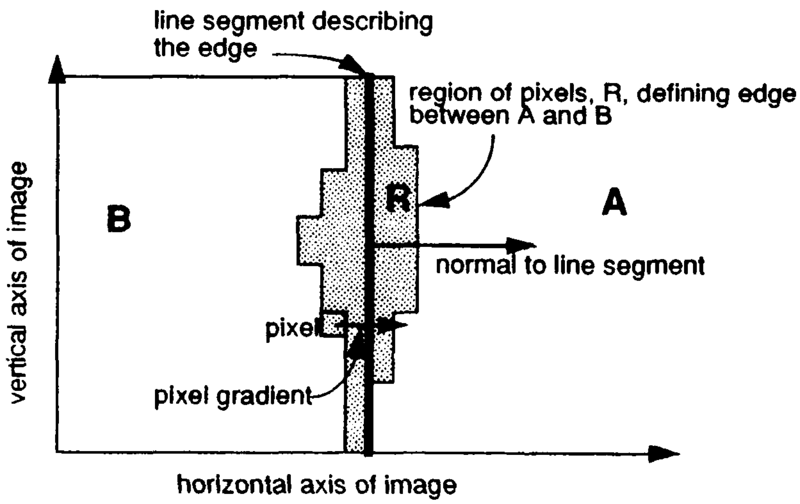

Figure 4.6. A region of pixels defining an edge in the gradient image. A and B are areas of common light intensity (pixels that are white in the gradient image). Area B is darker than A, subsequently all edge pixel gradients will be directed towards A. Similarly all edge pixel gradient angles will be close to the normal of the line segment describing the edge.

For two pixels to be of the same edge, they must both have gradient angles that are *close*. "Closeness" can be defined by ensuring the difference between two angles is less than some constant angle, C_2. The pixel gradient orientation is the normal to the tangent of the border between the light and dark areas of the image at that point (areas A and B in Figure 4.6). Therefore, all pixel gradient orientations will be close to the normal of the line segment describing the edge region. If a pixel's gradient angle is not close, then the pixel can not be considered to be part of that edge. The effects of varying C_2 are shown in Chapter VII.

The adjacency requirement can readily be seen in Figure 4.6. For a contiguous region of pixels to define an edge, all pixels in the edge must be adjacent to at least one other pixel in the edge. Thus, after testing a pixel's gradient magnitude for description of the gradient image, only the pixel's gradient angle and the pixel's adjacency with a previously edge-defined pixel needs to be considered (Figure 4.7). If these requirements fail, the pixel must still be considered being a member of an edge and therefore it will be the first pixel to define a new edge.

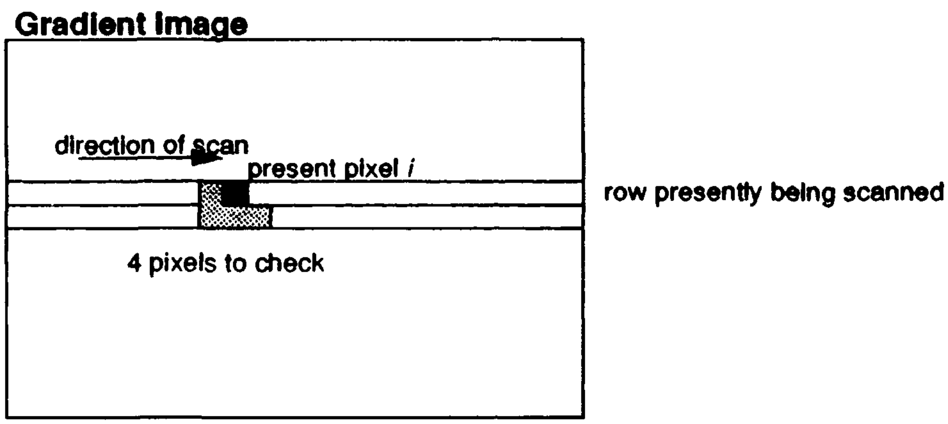

Figure 4.7. Pixel *i* is only compared with four adjacent pixels to check gradient orientation *closeness*. One of these four pixels must already belong to a defined edge.

C. DECIDING WHEN TO STOP ADDING PIXELS TO AN EDGE

Continually declaring new edges for every pixel that does not meet the [illegible]bership requirements for a declared edge will rapidly decrease performance as the number of edges gets large. It is desirable to process edges for linear fitting once it has been determined that no more pixels could possibly be included. This is done by enforcing the eight-pixel adjacency rule for pixel inclusion in an edge. Every time that the scan of pixels starts at a new row in the image, all edges must be inspected to ensure that the last pixel added to each was from the row just scanned (Figure 4.8). If this check fails, th[illegible] found on the present row (the row to be scanned) could be included with that edge, there[illegible] can be considered completed for grouping of contiguous pixels and may be processed for linear fitting. Thus all edges that pass this test are *active* edges that can still accept more pixels for inclusion.

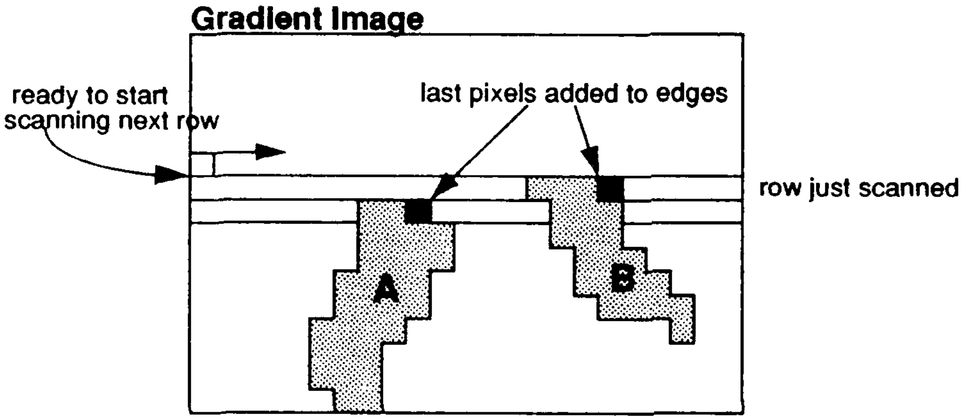

Figure 4.8. Prior to starting the scan of a new row, the list of edge regions must be reviewed to ensure that the last pixel added to each edge occurred in the row just scanned. If not, pixels found on the new row could not possibly be associated with the edge. Thus, edge A would be removed from the edge list whereas edge B is retained as an "active" edge that could still accept more pixels for inclusion.

V. LINE FITTING BY LEAST SQUARES

A. LEAST SQUARES FIT METHOD

One method for determine a line segment that adequately describes a two-dimensional region of datapoints is by least squares. Linear fitting by least squares is a simple and efficient method for calculating thousands of datapoints. In an image, the *least squares fit moments*, described by Kanayama [KAN90b], of pixel locations (x,y pixel coordinates of the image) for all pixels in the edge region will yield an associated line segment for the region's major axis. Verification of the region is required to ensure that the least squares fit moments are representative of a line segment. Regions that meet the requirements for a line segment are saved for the desired output and all others are discarded.

Once the decision has been made to include a pixel as a member of an edge region, the least squares fit moments must be updated for the region to include the new pixel. Consider the image as a cartesian coordinate system, discretely numbered by pixels in both horizontal and vertical directions. Each edge region, R, can be considered as a collection of pixels occupying a two-dimensional space (Figure 4.6). Pixels' positions (numbered by pixels in x and y axes) are used to define the least square fit moments m_{jk}.

1. Primary Least Squares Fit Moments

Each moment m_{jk} ($0 \le j,\ k \le 2$ and $j + k \le 2$) of an edge region, R, is defined as:

$$m_{jk} = \sum_{i=1}^{n} x^j \cdot y^k. \qquad (5.1)$$

Considering that m_{00} is equal to n, m_{00} defines the number of pixels associated with region, R.

The centroid C of an edge region, R, is defined as

$$C = \left(\frac{m_{10}}{m_{\text{[illegible]}}}, m\ \text{[illegible]}\right) \qquad (5.2)$$

2. Secondary L[illegible]nts

The sec[illegible] about the centroid are:

$$M_{20} \equiv \sum_{i=1}^{n} (x_i - \mu_x)^2 = m_{20} - \frac{m_{01}^2}{m_{00}}, \qquad (5.3)$$

$$M_{11} \equiv \sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_x) = m_{11} - \frac{m_{10}m_{01}}{m_{00}}. \tag{5.4}$$

$$M_{02} \equiv \sum_{i=1}^{n} (y_i - \mu_x)^2 = m_{02} - \frac{m_{01}^2}{m_{00}}. \tag{5.5}$$

3. Parametric Line Fitting of the Principle Axis

Using a parametric representation (r,α) of a line L, a point at coordinates (x,y) satisfies the equation

$$r = x\cos\alpha + y\sin\alpha \qquad \left(\frac{-\pi}{2} < \alpha < \frac{\pi}{2}\right). \tag{5.6}$$

The *residual* of pixel i (located at (x_i, y_i)) and the line L, defined by δ_i, is the distance perpendicular to L and pixel i.

$$\delta_i = (\mu_x - x_i)\cos\alpha + (\mu_x - y_i)\sin\alpha. \tag{5.7}$$

The *projection* of pixel i, p'_i, onto the line L is

$$p_i' = (x_i + \delta_i\cos\alpha, y_i + \delta_i\sin\alpha). \tag{5.8}$$

Therefore, p'_1 and p'_n, the projections of the first and last pixels associated with the edge region R onto the line segment L, can be used as estimates of the endpoints of L.

The sum of all the residuals is then

$$S = \sum_{i=1}^{n} (r - x_i\cos\alpha - y_i\cos\alpha)^2. \tag{5.9}$$

The line that best fits the set of points will minimize the sum S. Thus, the optimum line L (r,α) must satisfy

$$\frac{dS}{dr} = \frac{dS}{d\alpha} = 0. \tag{5.10}$$

Therefore,

$$\frac{dS}{dr} = 2\sum_{i=1}^{n}(r - x_i\cos\alpha - y_i\sin\alpha)$$

$$= 2\left[\left(r\sum_{i=1}^{n}1\right) - \left(\sum_{i=1}^{n}x_i\right)\cos\alpha - \left(\sum_{i=1}^{n}y_i\right)\sin\alpha\right]$$

$$= 2\,(rm_{00} - m_{10}\cos\alpha - m_{01}\sin\alpha)$$

$$= 0 \qquad . \tag{5.11}$$

and r can be expressed as

$$r = \frac{m_{10}}{m_{00}}\cos\alpha + \frac{m_{01}}{m_{00}}\sin\alpha = \mu_x\cos\alpha + \mu_y\sin\alpha, \tag{5.12}$$

where r may be negative. Substituting r in equation 5.9 by equation 5.12,

$$S = \sum_{i=1}^{n}(\,(x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha)^2. \tag{5.13}$$

Finally, $dS/d\alpha$ can be solved for.

$$\frac{dS}{d\alpha} = \sum_{i=1}^{n}(\,(x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha)\cdot(-(x_i - \mu_x)\sin\alpha + (y_i - \mu_y)\cos\alpha)$$

$$= \begin{matrix}\displaystyle\sum_{i=1}^{n}(\,(y_1 - \mu_y)^2 - (x_i - \mu_x)^2)\sin\alpha\cos\alpha \\ \displaystyle + \sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y)(\,(\cos\alpha)^2 - (\sin\alpha)^2)\end{matrix}$$

$$= \frac{1}{2}(M_{02} - M_{20})\sin2\alpha + M_{11}\cos2\alpha = 0 \tag{5.14}$$

Which provides a solution to the normal of the line L, α.

$$\alpha = \frac{1}{2}\operatorname{atan}\left(\frac{2M_{11}}{M_{02} - M_{20}}\right). \tag{5.15}$$

Equations 5.12 and 5.15 are the solutions for the line parameters generated by a least squares fit.

B. TESTS FOR VALIDITY OF AN EDGE REPRESENTING A LINE SEGMENT

The *equivalent ellipse of inertia* (Figure 5.1) for the edge region, R, will have the same moments about the centroid (M_{20}, M_{11}, and M_{02}) as R. M_{major} and M_{minor}, the moments about the major and minor axes of the ellipse, are defined as:

$$M_{major} = \frac{(M_{20}+M_{02})}{2} - \sqrt{\frac{(M_{02}-M_{20})^2}{4} + M_{11}^2}, \tag{5.16}$$

$$M_{minor} = \frac{(M_{20}+M_{02})}{2} + \sqrt{\frac{(M_{02}-M_{20})^2}{4} + M_{11}^2}. \tag{5.17}$$

The lengths of the major and minor axes, d_{major} and d_{minor}, are defined to be:

$$d_{major} = 4\sqrt{\frac{M_{major}}{m_{00}}}, \tag{5.18}$$

$$d_{minor} = 4\sqrt{\frac{M_{minor}}{m_{00}}}. \tag{5.19}$$

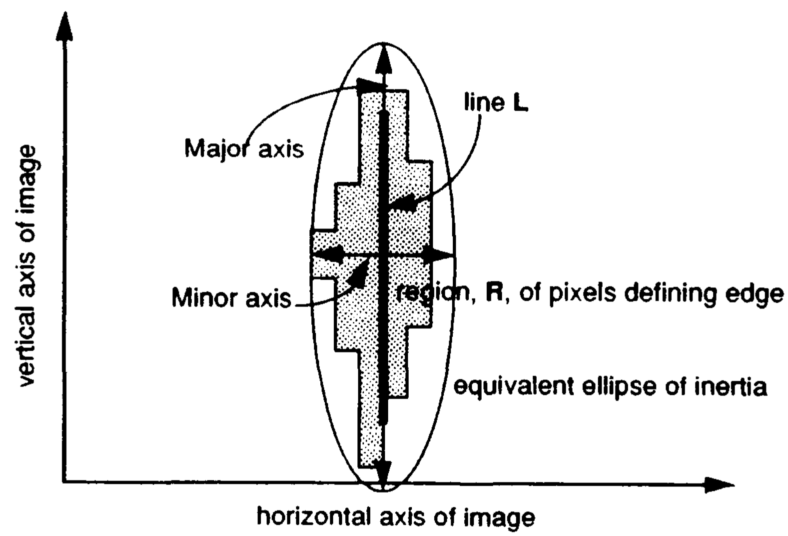

Figure 5.1. The major and minor axes of the equivalent ellipse of inertia for the region of pixels defining an edge.

For this implementation of least squares fit, the major axis will be considered to be the associated line segment for the edge region. The endpoints of the major axis will be the endpoints for the desired line segment. The projections of the first and last pixels, p'_1 and p'_n, associated with the region R, will therefore be used as estimates for the two endpoints of L. A ratio, ρ, of the axes length can then be used to describe the *thinness* of the ellipse.

$$\rho = \frac{d_{minor}}{d_{major}}. \tag{5.20}$$

Using the values for number of pixels (m_{00}), major axis length (d_{major}), and ellipse thinness (ρ) as parameters for comparison to edge significance, edge length and how much the edge resembles a line can be simply tested as long as the least squares fits moments are maintained for every pixel included in an edge. Let C_3 $(0 \leq C_3 \leq 1)$ be a constant for the maximum allowable ratio ρ that can be used to described a line. Let C_4 be a specified constant for the minimum number of pixels and C_5 for the minimum line length. Three requirements can therefore be specified for the line testing of an edge region.

1. the ratio of axes lengths less than the maximum ratio ($\rho < C_3$),
2. the number of pixels greater than the specified minimum ($m_{00} > C_4$), and
3. the line length of the region greater than the minimum length ($d_{major} > C_5$).

The ratio ρ proves to be the most significant measurement. For ρ to equal 1.0 means that the length of the minor axis is equal to the length of the major axis, representing an edge region that resembles a circular blob. Therefore, ρ can be compared to a maximum ratio, C_3, specified by the user. C_3 equal to 1.0 allows all edge regions to be considered thin enough to represent a line segment. A value of 0.1 seems to work well.

The two other tests can be used to trim down the number of smaller, less significant edges found. Specifying minimum number of pixels in an edge and minimum edge length, C_4 and C_5 respectively, is an effective means to filter out less significant line segments. The regions [illegible] meet these requirements will be saved for the desired output as the found line segments and all o[illegible]ns will be discarded. The effects of varying the parameters C_3, C_4, and C_5 are described in Ch[illegible]

VI. ALGORITHM FOR EDGE EXTRACTION

From the principles stated in the previous chapters, the following elements were identified to be incorporated the edge extraction algorithm:

1. Determine which pixels compose the edges.
2. Group pixels together into regions describing individual edges.
3. Describe a completed edge region in terms of a line segment via least squares fit of pixels' positions.
4. Ensure that line segment representation of the edge meets the specified thinness and length requirements.

The implementations are coded in the ANSI version of the C language and are contained in Appendix B. A user's guide for compiling and executing the implementations is in Appendix D.

A. THE *FINDEDGE* METHOD

Findedge is a C-program that follows the above outline. Remembering that the grayscale value for pixel *i+xsize+1* must be determined prior to the gradient calculation for pixel *i*, the algorithm for *findedge* can be stated to build both the grayscale and gradient images in one pass of scanning the input (color) image. This algorithm will ensure that gradient values are not calculated for the outermost pixels. Pixels with gradient magnitudes greater than the specified threshold, C_1, will be considered as datapoints for the edges.

Two tests were identified for a an edge-related pixel to be included as a member of a particular edge. Those tests were:

1. The pixel must be adjacent to a pixel already included in an edge.
2. The pixel's gradient angle must be "close" to that of the averaged angle of adjacent pixel's.

Two angles are "close" if the difference between them is less than a maximum constant, C_2. These tests can be conducted immediately once a pixel has been identified to be an edge-related pixel. Therefore, all black pixels in the gradient image will be tested for edge membership.

Pixels meeting the edge membership requirements must then be incorporated with the edge by updating the least squares fit moments, m_{jk}, for the edge region, R. This is accomplished by determining the *i*th pixel's two-dimensional image position (x,y) and summing the moments, m_{jk}, with the corresponding weights. If a pixel fails the membership test for the declared edges, then that pixel will be used as the first to declare a new edge. Thus, a linked list is the preferable data structure to maintain the declared edge regions.

Observing that new edges are declared as a linear scan of the image is conducted, and that the first requirement for pixel membership with an edge is adjacency, a pixel needs only to conduct the membership

test with the pixel to the immediate left (assuming a left-to-right, bottom-to-top scan). This builds edge regions for the present row only. Prior to continuing the scan at the start of the next row, the list of edge regions for the row just scanned will be compared with the list of edge regions from the previous row to see if they can be combined based upon vertical adjacency of edge regions between the two rows and "closeness" of average orientation of pixels' gradient angles.

Maintaining values for the first and last pixels included for each edge is necessary for the determination of line segment endpoints. Keeping track of first and last pixels added is straightforward with only one exception for horizontally oriented lines. The case for horizontal lines (Figure 6.1) requires determination of endpoints based upon the leftmost and rightmost pixels of the edge vice the first and last pixels added.

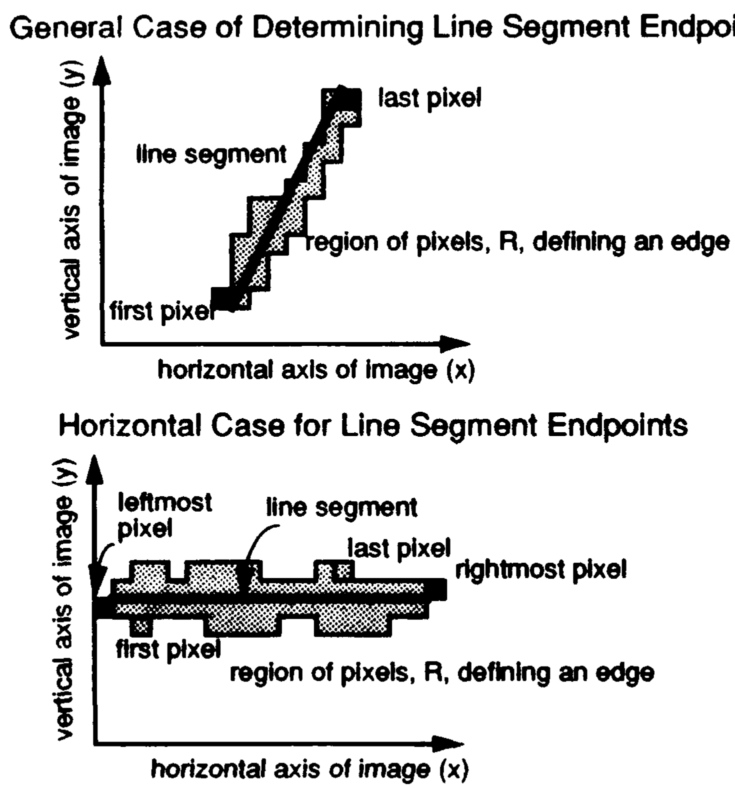

Figure 6.1. General and horizontal cases for determining pixels to be used for calculation of line segment endpoints. The first and last pixels in the horizontal case may not necessarily fully describe the extent of the associated line segment. in which case the leftmost and rightmost pixels should be used.

Edges from the previous row that are not appended with pixels from the row just scanned are considered finished with their construction and can therefore be tested for meeting the three requirements for a line segment. These requirements were identified to be:

1. The ratio of lengths of major and minor axes (ρ) < C_3.
2. Number of pixels in an edge (m_{00}) > C_4.
3. The length of the major axis (d_{major}) > C_5.

Determining d_{major} and ρ is easily done since the primary least square fit moments are updated for every pixel added. Edges that meet these requirements will serve as the desired output of straight edges, otherwise they are discarded and the associated memory is freed.

B. AN ATTEMPT TO CUT COMPUTATION COSTS

The *findedge* algorithm requires an averaged value of the pixel gradient angle orientations to be maintained for each edge. This average value is used for the comparison with pixel gradient orientation to determine angle "closeness." Thus, for every pixel added to an edge, the average of the pixels' gradient orientation must be calculated. To alleviate these computations, the specified maximum value for angle closeness, C_2, could be used to determine two *static* sets of angles that any pixel gradient orientation could be defined by. These two sets could be thought of as dividing up all possible angles between π and $-\pi$ by C_2 as illustrated in Figure 6.2.

"Closeness" could now be defined by comparing the two groups that a pixel gradient angle belongs to with the groups of the adjacent edge-related pixel's gradient angle. If either groups are similar, then the gradient angles are close. This method is termed *static* since the two sets of angles are computed only once and remain unchanged whereas the *findedge* method continually recomputed the average gradient angle of the pixels in the edge region. The results of this method of pixel grouping into straight line edges are shown in Chapter VII. Compared with the line segments from the *findedge* algorithm, more line segments are extracted but shorter and more fragmented. Since the desired output is a simple structure (line segments) that best represents the linear features in an image, the performance of the *static* method is worse.

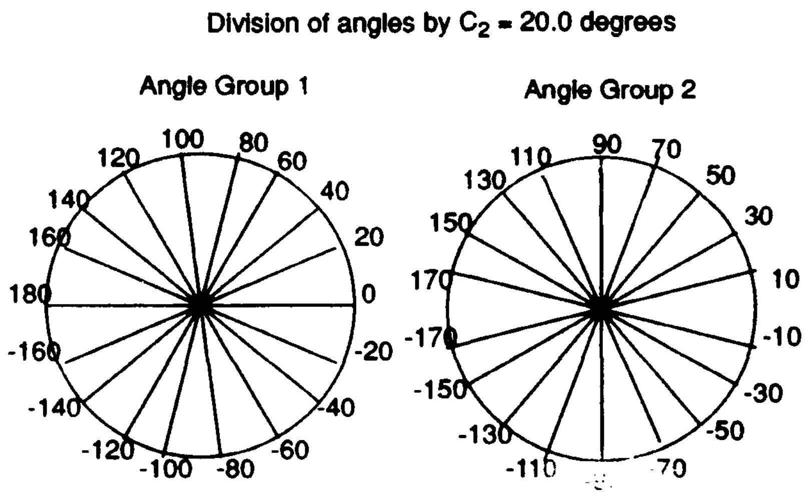

Figure 6.2. The [illegible] groups of angles determined for C_2 = 20.0 degrees. Each pixel gradient orientation will belong to one region from each group of angles. The groups can be thought of as overlapping in such a way so that "close" gradient angles not belonging to the same region in one group will belong to the same region in the other group.

C. IGNORING GRAYSCALE CONVERSION WITH THE *FASTEDGE* METHOD

Computation costs can still be reduced in the *findedge* algorithm by omitting the construction of the grayscale and gradient images. Observing that the green component of the RGB color format accounts for the majority of grayscale weights, pixel gradients can be estimated by using only the green component value. This is accomplished with a bitmask to get only the green color values for use with the Sobel operator (equations 4.3 and 4.4). Thus, construction of the grayscale image is unnecessary. Similarly, by observing that the gradient image only identifies those pixels that describe [illegible] are not required for the construction of the edges, its construction and storage as a real image will be omitted.

Therefore, the program *fastedge* can perform the same straight line edge extraction as *findedge*; however, since the values for pixel gradient calculation are only the green color components and not the weighted grayscale sum, a new gradient magnitude threshold value (C_1) must be determined. The *fastedge* implementation displays the input (color) image and draws the extracted lines over the image. Results are in Chapter VII. A guide for compiling and using *fastedge* is contained in Appendix D.

VII. IMPLEMENTATION

A. SYSTEM DESCRIPTION

The graphics workstation used for feature extraction is a SiliconGraphics Personal Iris. Its graphics support libraries are implemented in the C language; therefore, the code for the *findedge* and *fastedge* methods is in C. The VideoFramer frame grabbing hardware is a daughterboard on the Iris that accepts the pixel information after it has been converted from the video source signal to an acceptable pixel format (e.g. RGB). A typical image from the charged-couple device (CCD) television camera is 646 x 486 pixels. That is 313,956 total pixels to be stored in a file approximately 1.6 Mbytes. Efficient algorithm design is paramount for such large image files. The Personal Iris is a 35 MHz machine that can read an image file and conduct edge extraction via the *fastedge* implementation in approximately 15 seconds (about 3 seconds for an image "shrunken" to half of its dimensions).

B. THE INPUT IMAGE

Prior to encoding an algorithm to find the straight line edges in an image, a careful analysis of the image must be considered. The CCD camera used (JVC model TK870U) can provide a video signal in various formats. Some of these video formats include:

- Betamax Video
- D1 Video
- NTSC (National Television Systems Committee format)
- PAL
- RGB (red, green, blue attributes)
- Super VHS Video

There are also various formats for saving and displaying images as files on a computer. Some of these formats include:

- Apple Macintosh MacPaint file
- Compuserve Graphics Interchange Format file (GIF)
- Encapsulated Postscript file (EPS)
- Hierarchical Data file
- PIXAR picture file
- Portable Bitmap file
- Postscript file
- SiliconGraphics Image RGB file (SGI)
- Stardent AVS X image file
- Sun Rasterfile
- Tagged image file (TIF)
- Utah Run-Length-Encoded image file
- Wavefront raster image file
- X11 bitmap file

The RGB format was chosen to be utilized because it is easily translated into NTSC (National Television Standard) and exists as a standard format for both the CCD camera output and the SiliconGraphics workstation.

1. The RGB Format

The RGB format is a 32 bit digital format to describe a single pixel in terms of its basic color components (red, green, and blue). In the C language, a 32 bit format is referred to as a "long integer" data type. Eight bits are used to represent each of the intensities for the red, green, and blue color components of the pixel. This allows for intensity values to range from 0 to 255 decimal (0 to FF hexadecimal) for each component. An intensity value of 0 represents zero light intensity for that color component, 255 for full illumination.

The RGB format uses the red, green, and blue colors as the basic components because they are the three primary *additive* colors. An additive color scheme used on television and similar displays combines colors from red, green, and blue light sources together to create the rest of the colors in the visual spectrum. This is contrasts with the *subtractive* color scheme implemented with yellow, cyan, and magenta as the three primary colors which is used in film photography for filtering a white light source to create the other colors.

The RGBA format only differs from the RGB format in that it has eight bits reserved for the *alpha* component which is used for *transparency* on graphics workstations. An alpha component with a value of zero is completely transparent, whereas a value of 255 is completely opaque. The RGB and RGBA formats are both 32 bits long and are basically identical. The RGB format simply ignores the alpha component and operates as if all colors are opaque (alpha value set to FF hexadecimal). Figure 7.1 shows the bit placement for the RGB/RGBA formats. In order to allow easier translation between these two formats, the alpha component is set to FF hexadecimal. To illustrate this format with some examples, the color black is represented as FF000000 hexadecimal (alpha = FF; blue, green, red = 00), white is FFFFFFFF, and bright kelly green is FF00FF00.

	alpha	blue	green	red
bits	31	23	15	7 — 0

Figure 7.1. The RGB / RGBA formats. RGB format ignores the alpha block.

2. Color to Grayscale Conversion

The *findedge* implementation of feature extraction requires a grayscale image. However, if a color camera is used, it will be necessary to convert the color image to a grayscale image. The RGB format describes each pixel in terms of its basic additive primary colors, whereas the overall intensity of the attributes would be a grayscale value. To determine the grayscale value, each of the color values must be weighted and summed. The weights assigned to each color for representation as a black and white television (grayscale) pixel have been standardized by the National Television Systems Committee. The following weights are prescribed by the NTSC [HAL89].

Red = 0.299
Green = 0.587
Blue = 0.114

Converting a color pixel to a grayscale value is simply assigning the sum of the weighted color component values to the grayscale value (equation 7.1).

$$RGB_{grayscale} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} . \qquad \text{(Eq 7.1)}$$

4. The NPSIMAGE Data Structure

The RGB format is a well known and widely used format for defining a single color pixel. However, formats for describing entire images via RGB pixels can still vary. One such format, a simplified model of a standard video image, developed by Mike Zyda at the Naval Postgraduate School, is the NPSIMAGE data structure outlined in the *image_types.h* file in Appendix A. NPSIMAGE is a C language data structure (a record type) to store an RGB, RGBA, or color-mapped image.

For all video images, the pixels are represented as a single, large, one-dimensional array of values starting with the first pixel in one corner of the two-dimensional image and ending at the opposite corner (Figure 7.2). All pixels in this array are arranged via a single, linear scan of the video image (i.e. left to right, bottom to top). In order to properly display the image from a saved format, it is crucial to have the dimensions of the original image saved. The dimensions of the input image are expressed in number of pixels in the horizontal and vertical directions. NPSIMAGE has two integer value slots, `xsize` and `ysize`, to store these dimensions.

The image format must subsequently provide an efficient way to access the one-dimensional array of pixel values. This is generally done in the C language via a pointer initialized to the first pixel of the one-dimensional array. Moving the pointer by altering its offset allows easy access to all pixel values. When using the NPSIMAGE data structure, such a pointer will be initialized to the `imgdata.bitsptr` long integer array (Figure 7.3).

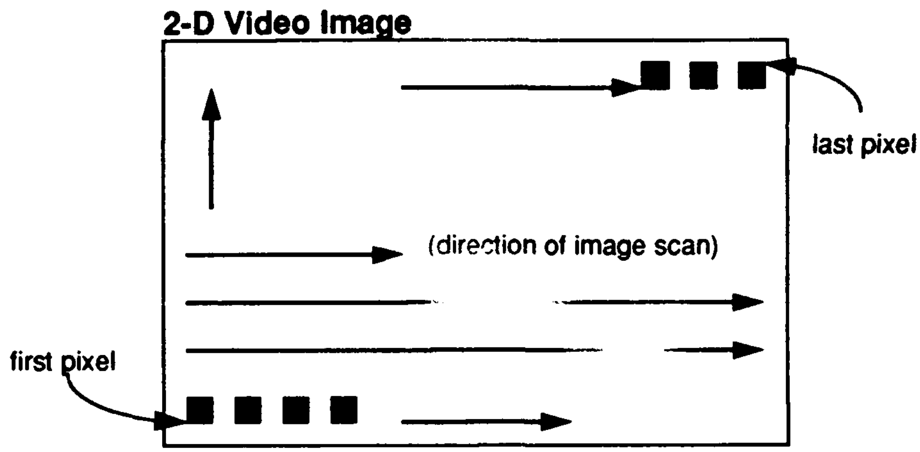

Figure 7.2. Construction of one-dimensional array of pixel values from two-dimensional video image. Order of pixel scan is left to right, bottom to top, as used in the NPSIMAGE data structure.

```
NPSIMAGE *img;    /* a pointer to the image data structure */

long *ptr;    /* a pointer to pixel values
              (represented as long integers) */

ptr = img->imgdata.bitsptr;
```

Figure 7.3. Declaration and initialization of a C [illegible] pointe[illegible] the RGB values in an NPSIMAGE.

VIII. EDGE EXTRACTION RESULTS

A. *FINDEDGE* IMPLEMENTATION

The implementation of the *findedge* algorithm described in Chapter VI and detailed in Appendix B provides extracted line-segment features such as in Figure 8.1. The line segments found are from the input image of the chair (Figure 3.3). The five parameters for gradient thresholds and line segment properties (C_1 through C_5) were determined by trial and error. The effects of modifying the parameters is seen in later in this Chapter.

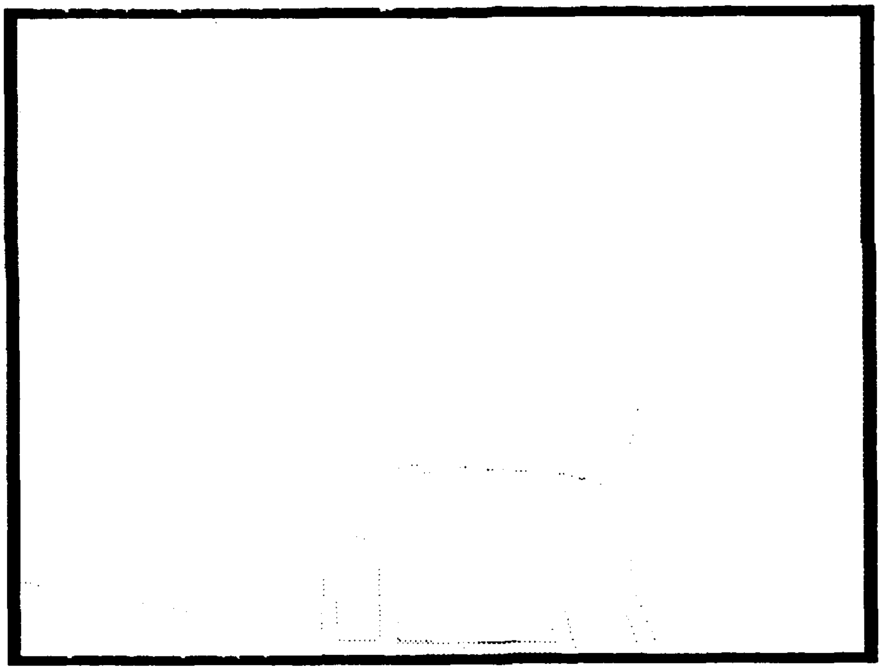

Figure 8.1. Lines found from image of chair via *findedge* implementation.
C_1 = 5,000,000.0; C_2 = 20.0 degrees; C_3 = 0.1; C_4 = 20 pixels; C_5 = 20.0.

B. *STATIC* METHOD OF GRADIENT ANGLE TESTING

Results from the *static* method for determination of "closeness" of pixel gradient angle orientation are displayed in Figures 8.2 through 8.4. The input image is that of the chair (Figure 3.3) and the resultant line segments can be compared with Figure 8.1. All five parameters are the same as those used in Figure 8.1. Figure 8.2 shows the lines found using the first group of static angles divided by 20 degrees (see Figure 6.2) and Figure 8.3 shows the lines from the second group of static angles. Figure 8.4 displays all lines found from both Figures 8.2 and 8.3. Comparing Figures 8.1 and 8.4 demonstrates the shortcomings of the *static* method. The line segments are shorter, broken, and subsequently more numerous, all of which are traits not desired for the purposes of pattern-matching.

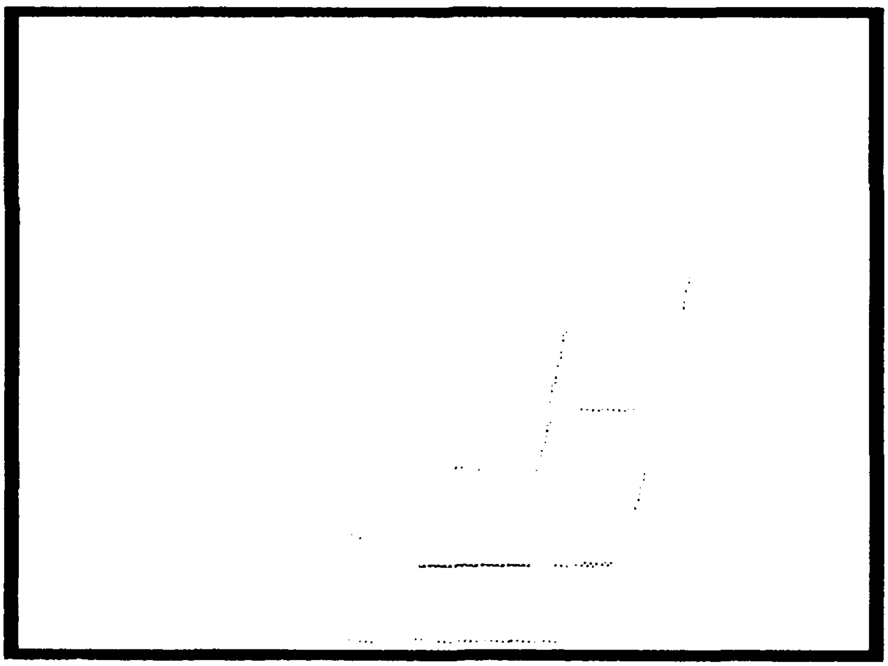

Figure 8.2. Lines found from image of chair via *static* gradient angles method (group 1 of Figure 6.2). C_1 = 5,000,000.0; C_2 = 20.0 degrees; C_3 = 0.1; C_4 = 20 pixels; C_5 = 20.0.

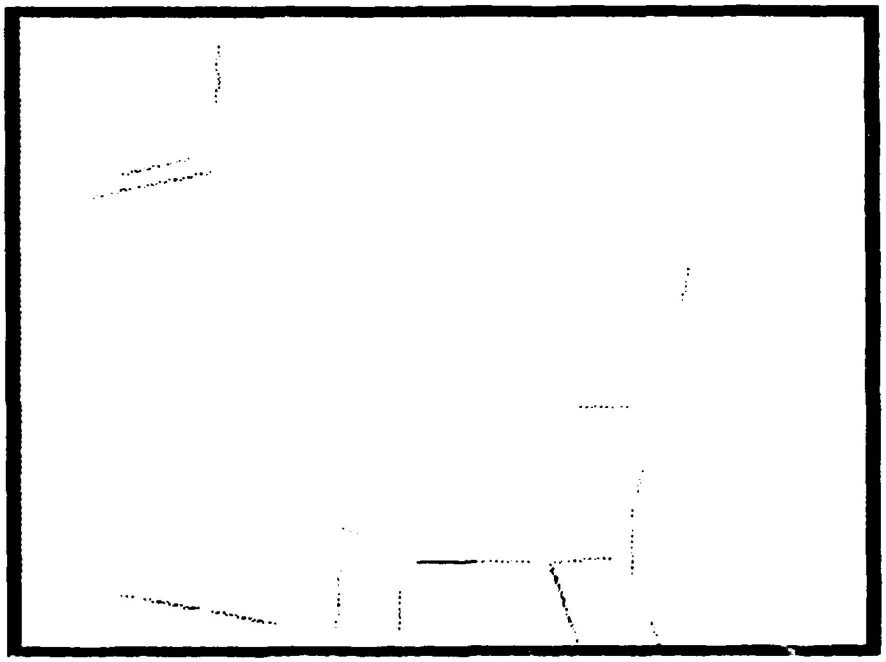

Figure 8.3. Lines found from image of chair via *static* gradient angles method (group2 of Figure 6.2). C1 = 5,000,000.0; C2 = 20.0 degrees; C3 = 0.1; C4 = 20 pixels; C5 = 20.0.

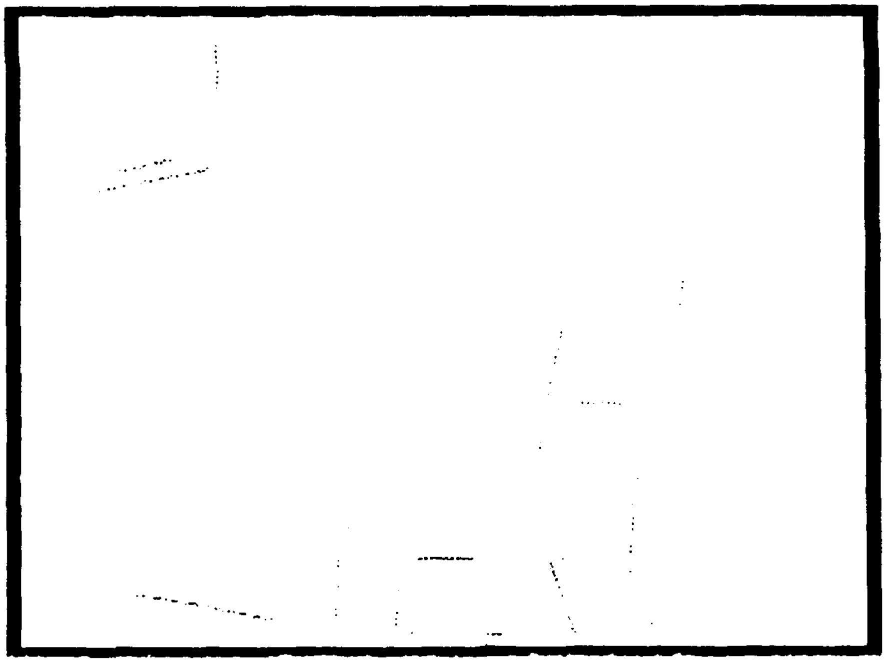

Figure 8.4. All lines found from image of chair via *static* gradient angles method. C_1 = 5.000.000.0; C_2 = 20.0 degrees; C_3 = 0.1; C_4 = 20 pixels; C_5 = 20.0.

C. *FASTEDGE* IMPLEMENTATION

The last algorithm described in Chapter VI, the *fastedge* method produced lines very similar to those found by the *findedge* method. Figure 8.5 shows the resultant line segments via the *fastedge* implementation. The major difference between this and *findedge* was determination of the gradient magnitude threshold value C_1. *Findedge* used color to grayscale conversion for an absolute intensity value whereas *fastedge* only used the green color component of the RGB image.

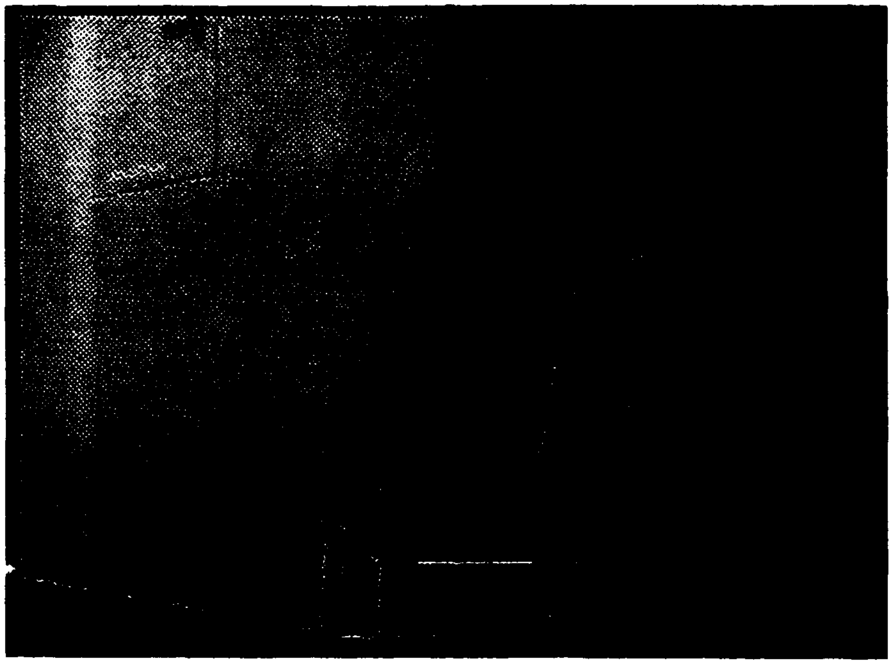

Figure 8.5. Lines found from image of chair via *fastedge* implementation.
$C_1 = 30{,}000.0$; $C_2 = 20.0$ degrees; $C_3 = 0.1$; $C_4 = 20$ pixels; $C_5 = 20.0$.

D. EFFECTS OF EDGE EXTRACTION AFTER "SHRINKING" AN IMAGE

Condensing an input image to half of its dimensions prior to performing the straight edge feature extraction routines produces some noteworthy results. Aside from requiring only a quarter of the computational time for edge extraction, the resultant line segments are generally fewer, longer, and seem to model the image in simpler terms. To illustrate such differences, Figure 8.6 is resultant line segments (via *fastedge*) of the chair image condensed to half of its length and width (quarter of the original area). Its results can be compared with the lines extracted in Figure 8.5. Similarly, Figure 8.7 is a view of a hallway interior with its extracted line segments (Figures 8.8 and 8.9) to be compared with the edges extracted from the same image after shrinking (Figure 8.10) and edge extraction (Figures 8.11 and 8.12).

Figure 8.6. Lines found from image of chair after it has been shrunken to half of its dimensions and using the *fastedge* program. $C_1 = 30{,}000.0$; $C_2 = 20.0$ degrees: $C_3 = 0.1$; $C_4 = 20$ pixels; $C_5 = 20.0$.

Figure 8.7. Image of hallway interior.

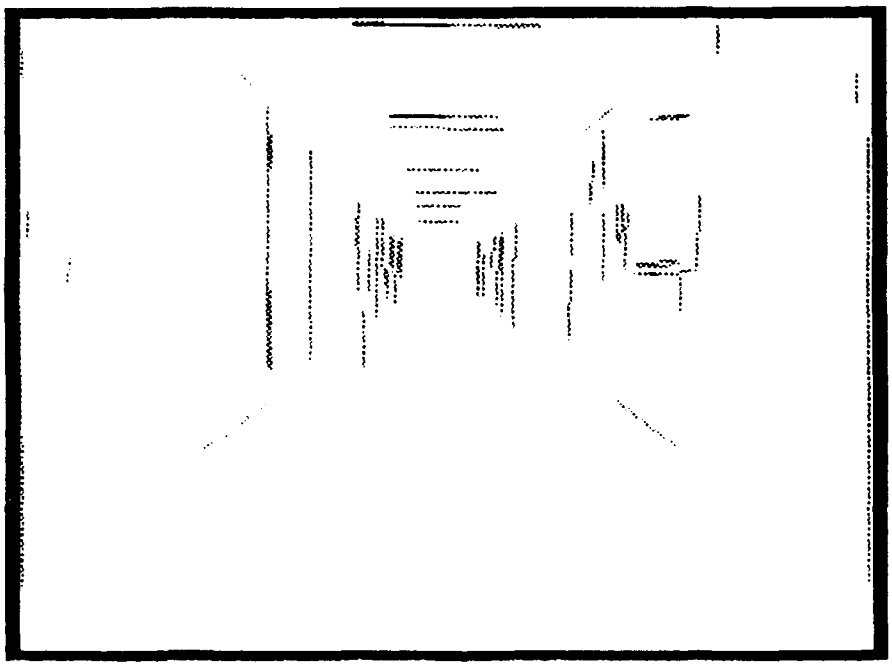

Figure 8.8. Extracted edges of hallway interior. Edge extraction by *fastedge* method. $C_1 = 30.000.0$; $C_2 = 20.0$ degrees: $C_3 = 0.1$; $C_4 = 20$ pixels; $C_5 = 20.0$.

Figure 8.9. Extracted edges superimposed over image of hallway interior.

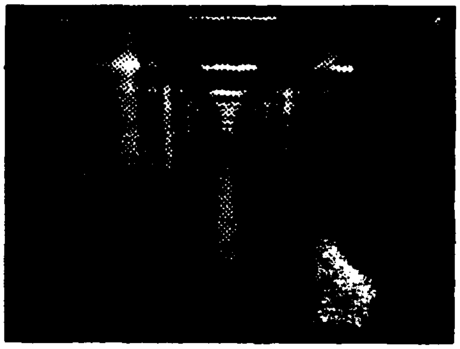

Figure 8.10. Shrunken image of hallway interior.

Figure 8.11. Extracted edges of shrunken h[illegible] by *fastedge* method.
$C_1 = 30{,}000.0$; $C_2 = 20.0$ degrees; C_3 = [illegible] = 20.0.

Figure 8.12. Extracted edges superimposed over image of hallway interior.

E. EFFECTS OF VARYING FEATURE EXTRACTION PARAMETERS

1. Gradient Magnitude Threshold C_1

The threshold value C_1 is used to determine if the gradient magnitude of a pixel is sufficiently large enough to describe an edge. If the value for pixel gradient magnitude, determined by equation 4.1, is larger than C_1, then the pixel in the gradient image is set black, otherwise it is set white. In the *findedge* program, the value for C_1 was determined to be 5,000,000.0 by testing. Figures 8.9 through 8.11 show the associated gradient images for values of C_1 equal to 5M, 8M, and 2M, respectively.

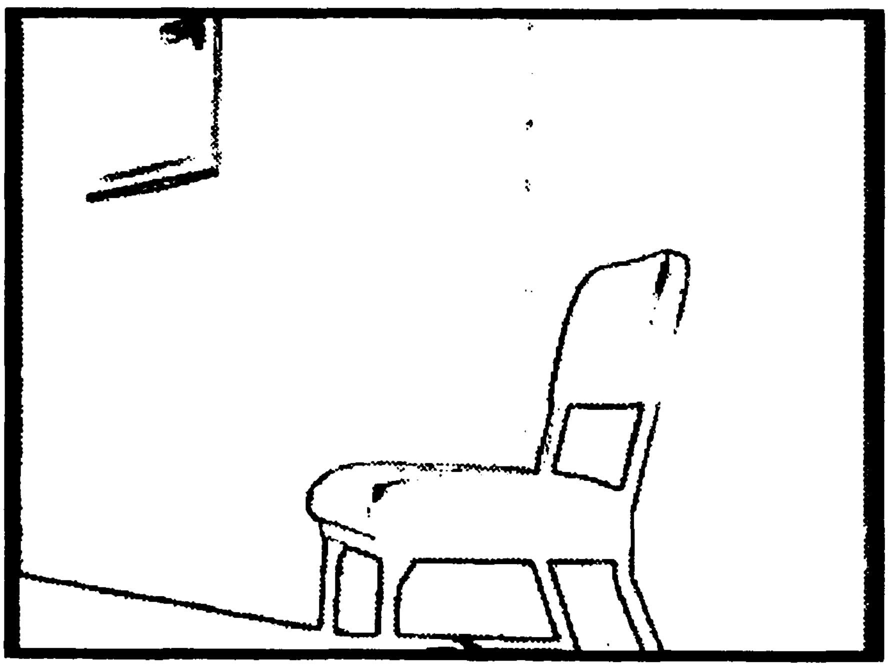

Figure 8.9. Gradient image of a chair with magnitude threshold C_1 = 5,000,000.0.

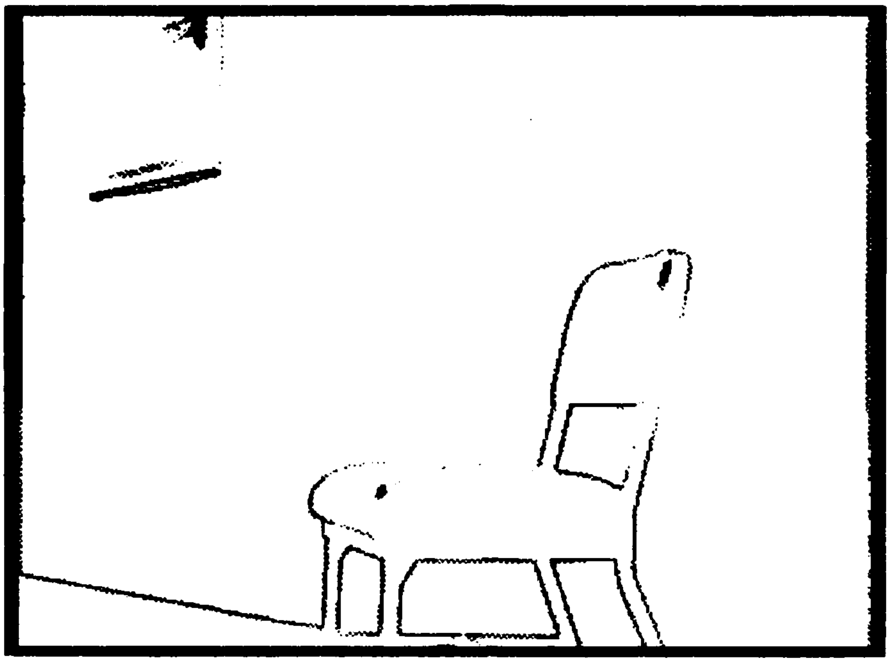

Figure 8.10. Gradient image of a chair with magnitude threshold C_1 = 8,000,000.0.

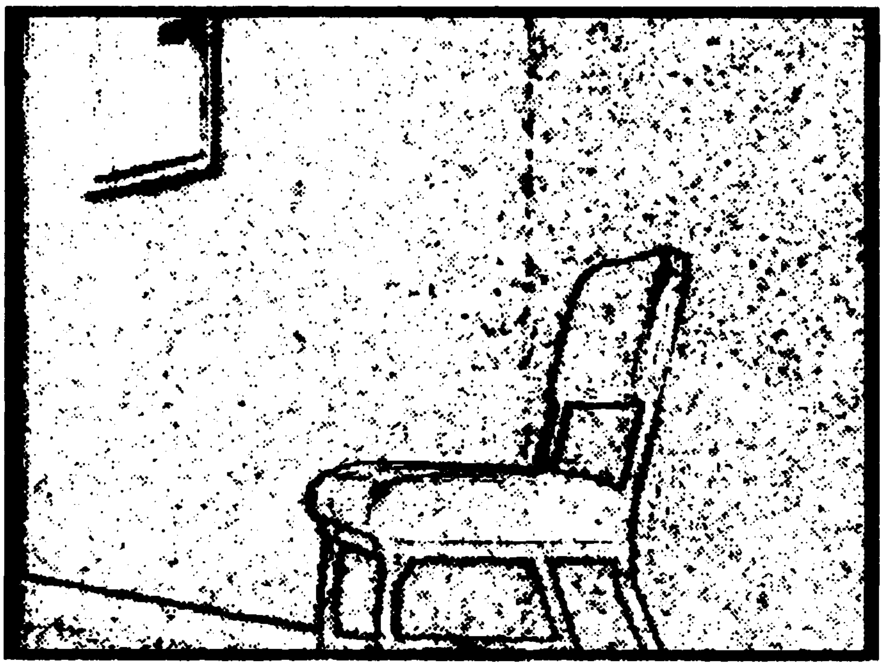

Figure 8.11. Gradient image of a chair with magnitude threshold C_1 = 2,000,000.0.

2. Gradient Angle "Closeness" Angle C_2

The threshold value C_2, the maximum difference between two pixel gradient orientations was tested by trial and error to produce more extracted line segments that best represented the straight edges from the input image. For the image of the chair (Figure 3.3), C_2 = 17.2 degrees performed best. Larger angles resulted in combining multiple straight edges from the image into one extracted line segment, whereas smaller angles were too restrictive to produce long line segments. The best C_2 value for one image is not necessarily the best value for all images. Pictures of the hallway interior, used for pattern-matching and pose-determination, utilize C_2 = 28.65 degrees (0.5 radians). The images in Figures 8.12 through 8.14 illustrate the effects of modifying C_2 of the chair image via the *fastedge* implementation.

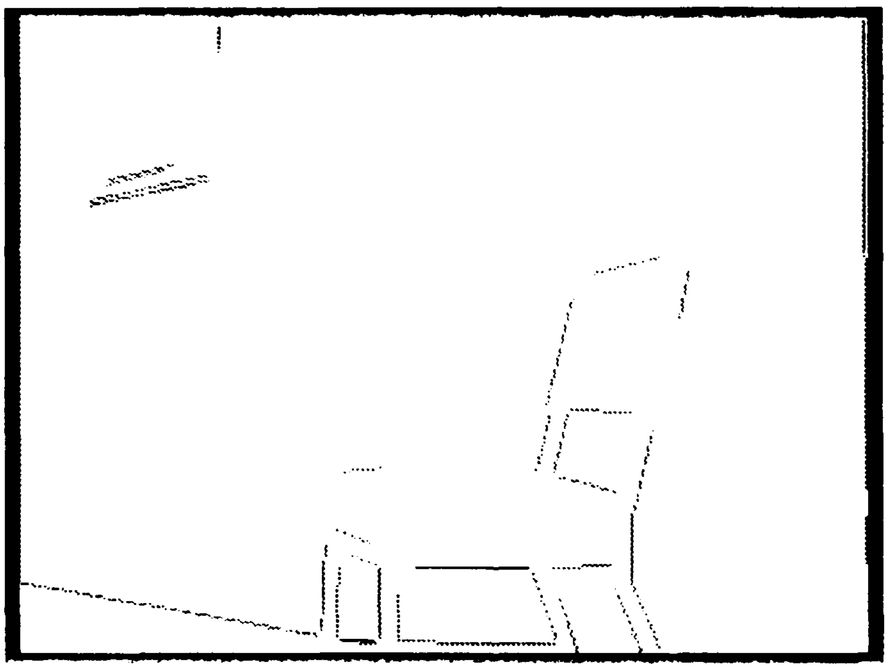

Figure 8.12. Lines extracted from image of chair for C_2 = 17.2 degrees (0.3 radians). C_1 = 30,000; C_3 = 0.1; C_4 = 20 pixels; C_5 = 20.0.

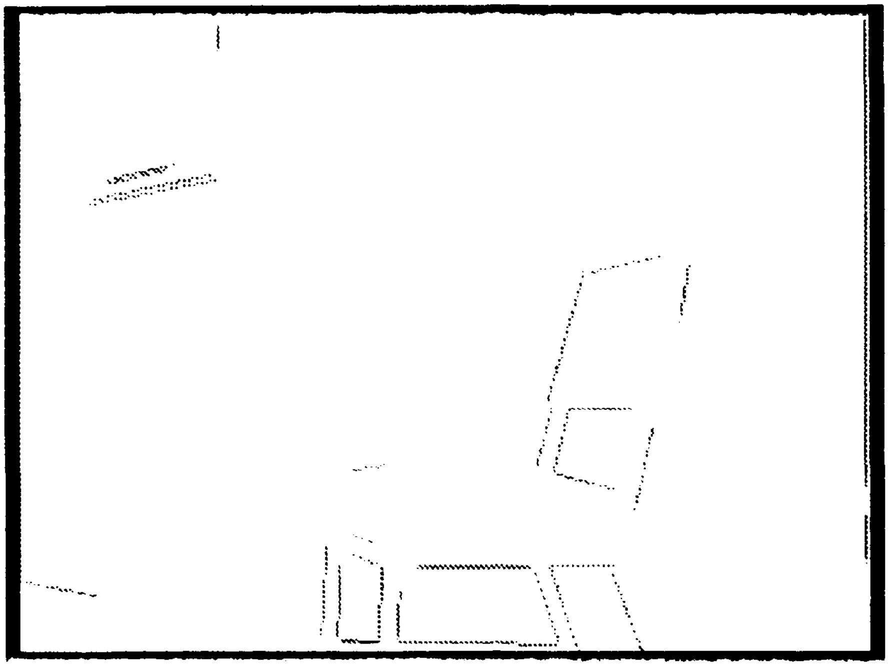

Figure 8.13. Lines extracted from image of chair for C_2 = 30 degrees.
C_1 = 30,000; C_3 = 0.1; C_4 = 20 pixels; C_5 = 20.0.

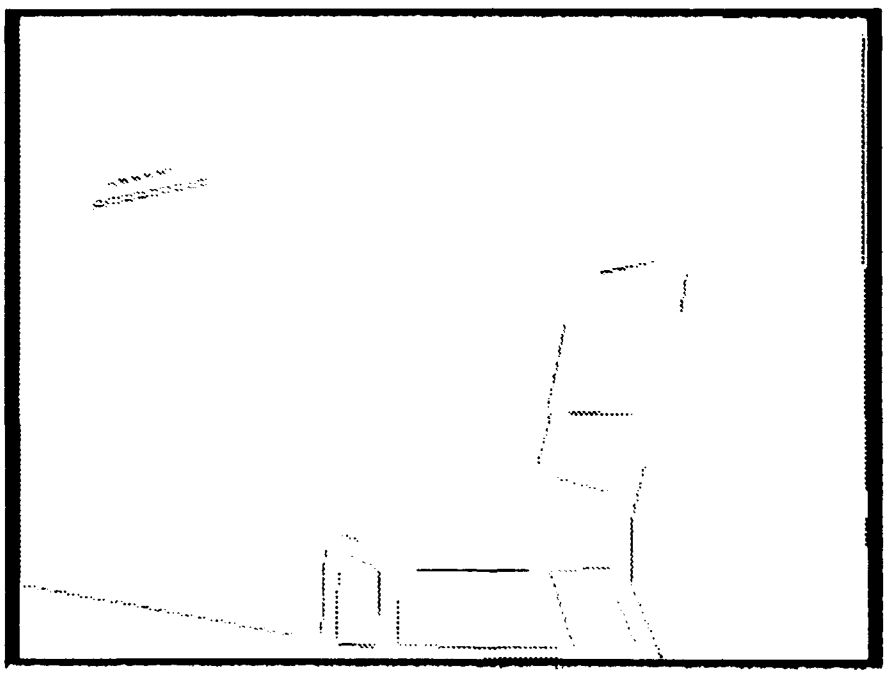

Figure 8.14. Lines extracted from image of chair for C_2 = 12 degrees.
C_1 = 30,000; C_3 = 0.1; C_4 = 20 pixels; C_5 = 20.0.

3. Line Test Parameters C_3, C_4, and C_5

The three line test parameters identified in Chapter V are utilized for limiting the number of extracted line segments by placing restrictions on the ratio of axes lengths ($\rho = d_{minor}/d_{major}$), the number of pixels (m_{00}), and the major axis length (d_{major}), respectively. The lines extracted from the chair image by modifying C_3, C_4, and C_5 are shown in Figures 8.15 through 8.17. Each Figure can be compared with the baseline extracted features in Figure 8.12.

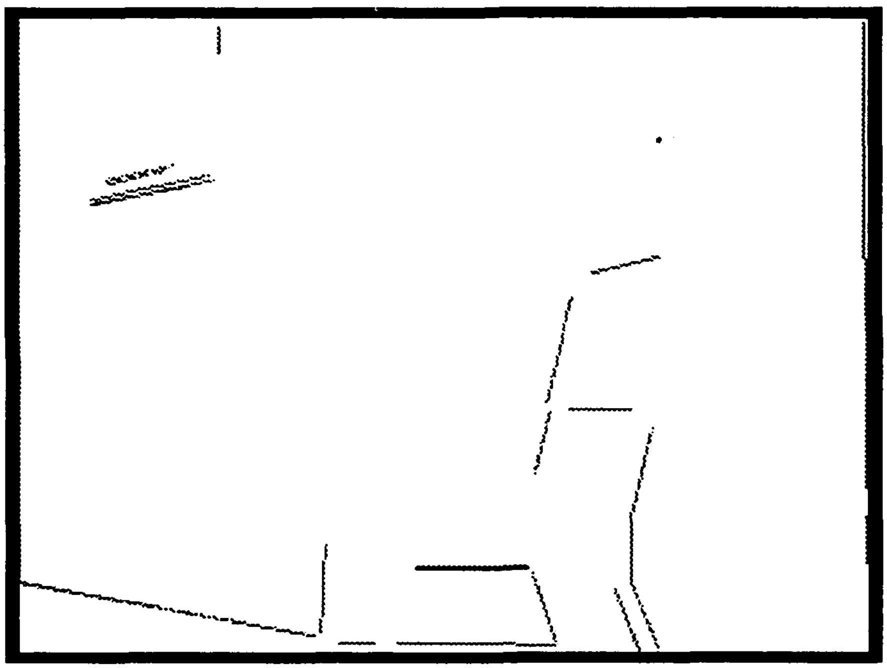

Figure 8.15. Lines extracted from image of chair for maximum thinness ratio $C_3 = 0.05$. $C_1 = 30{,}000$; $C_2 = 17.2$ degrees; $C_4 = 20$ pixels; $C_5 = 20.0$.

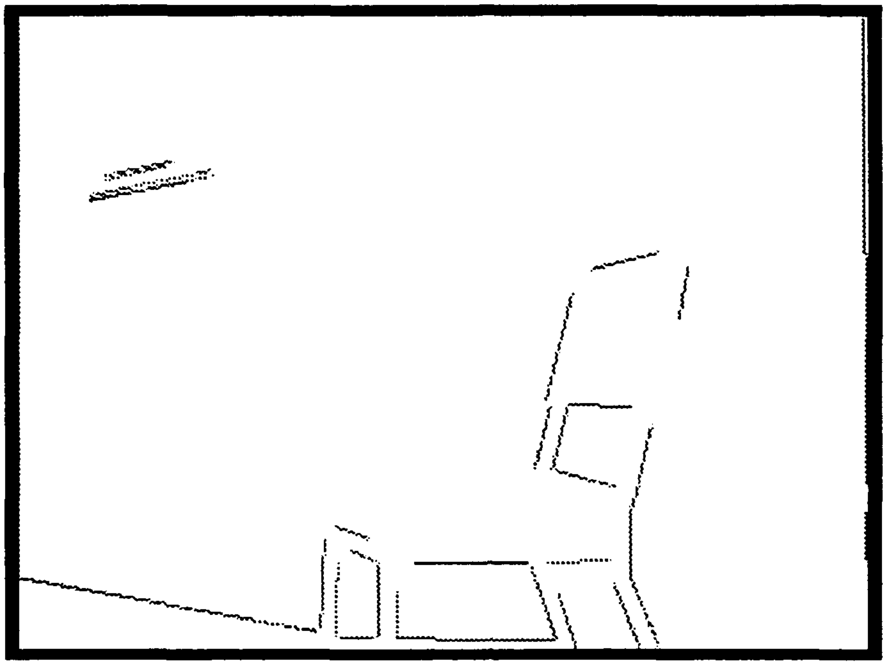

Figure 8.16. Lines extracted from image of chair for minimum number of pixels $C_4 = 40$ pixels. $C_1 = 30{,}000$; $C_2 = 17.2$ degrees; $C_3 = 0.1$; $C_5 = 20.0$.

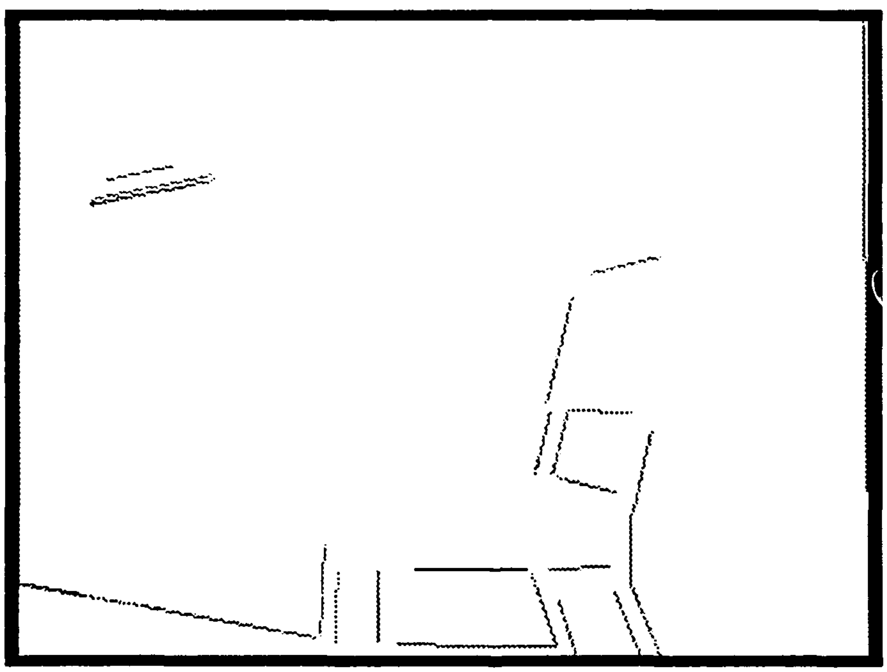

Figure 8.17. Lines extracted from image of chair for minimum axis length $C_5 = 50.0$. $C_1 = 30{,}000$; $C_2 = 17.2$ degrees; $C_3 = 0.1$; $C_4 = 20$ pixels.

IX. MATCHING LINE SEGMENTS

The focus of this Chapter is the development of a matching algorithm that compares the straight edges extracted from the input image with the linear features of the environment model. The pattern matching of two-dimensional line segments can be divided into two different problems:

1. matching with the model based upon a known location and orientation of the robot (the *pose* or *configuration* of the robot),
2. and matching with the model without knowledge of the robot's pose.

The first problem is applied to position correction and vehicular navigation, whereas the second problem is that of object identification and viewing aspect determination. The first problem is much more constrained than the second and is pursued herein towards the goal of pose determination and vehicle navigation. Expecting small dead-reckoning errors to arise in the vehicle's pose, the matching algorithm must exhibit some degree of robustness if it is to make valid corrections. Therefore, the goal of the matching process is to find the best match of image lines to model lines.

A. THE ENVIRONMENT MODEL

The testing and operating environment for Yamabico-11 is the fifth floor of Spanagel Hall at the Naval Postgraduate School. Modeling this environment in a graphical database [STE92] was accomplished in conjunction with the above edge extraction methods for the pattern matching task described in Chapter IX. The model is a "2-D +" model similar to that described by David Marr [MAR79]. The environment model is designed to support requirements for visual navigation, sonar navigation, shortest distance path determination, and safe path planning. The floorplan features are expressed in the *xy* cartesian plane and associated heights of hallway features are modelled in the *z* axis. The notion of *free space* is aided by defining polygons in the *xy* plane as either *floor* or *ceiling* polygons. All database measurements are in inches. All support routines for implementation of the model are coded in ANSI C and described fully in [STE92].

1. Interfacing the Model Database

Since the application features of the environment were encoded in the same language as the image processing and pattern matching methods, accessing the model is quite simple. The pattern matching routine will invoke calls to both the image processing and environment model functions to access the line segment primitives utilized in the matching process. The primary call to the model (*get_view()*) requires providing an estimated pose (position and orientation information) of the robotic vehicle and the focal length of the camera

lens. The function returns a data structure that includes pointers to linked lists of the line segments of the model database mapped to a two-dimensional viewing plane that the robot should expect to see given the input pose and visibility constraints of occluding edges. The line segments shown in the model's two-dimensional view represent the orthogonal features of the operating environment, primarily the junctures of the walls, ceiling, floor, doors, and overhead lights.

B. BASIC COMPARISON OF LINE SEGMENTS

Since the environment model provides a two-dimensional view of the known linear features for a given pose, the matching algorithm needs only to determine linear matches in two dimensions. From the results seen in Chapter VII, the extracted edges from the input image do not cross, overlap, or touch. The extracted linear edges are, at best, incomplete segments of the modelled linear features. The matching algorithm must be based upon individual matches between the image's extracted edges and the model's linear features. When comparing two-dimensional line segments, three aspects must be considered: translation, rotation, and scaling differences (Figures 9.1 through 9.3).

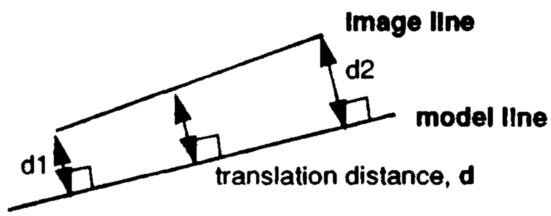

Figure 9.1. The translation distance, *d*, between two lines. Can be computed as the median between distances *d1* and *d2*, perpendicular to the model line.

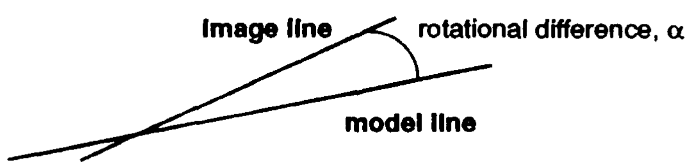

Figure 9.2. The rotational difference, α, between two lines.

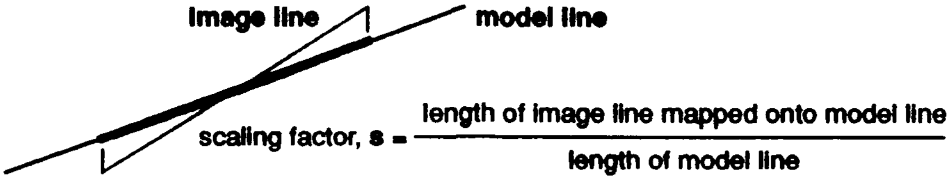

Figure 9.3. The scaling factor, s, between two lines. S is the ratio of the image line mapped onto the model line with the model line itself.

An additional requirement based upon the properties of the extracted edges using least squares fit can also be stipulated. That is, the endpoints of the image edges must lie within the endpoints of the modelled line segment. This check for *endpoint inclusion* may be relaxed to allow for vehicle translation errors. This is done by keeping the difference that the image edge endpoints lie outside of the endpoints of the modelled line segment, divided by the length of the image edge (d_{major}), less than or equal to a specified amount (k in Figure 9.4).

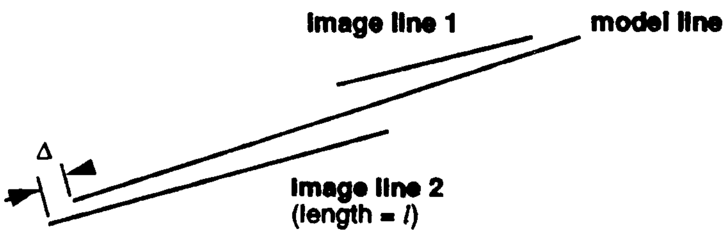

Figure 9.4. The inclusion of image endpoints within the model line's endpoints. Image line 2 can meet this requirement if $(\Delta / l) < k$, $(0.0 < k < 0.1)$.

Another consideration for matching image lines to model lines is that since the extracted image lines are often incomplete and broken segments of a modelled feature, many image lines may be matched to one model line as shown in Figure 9.5. The only stipulation is that the image lines do not overlap when mapped onto the same model line.

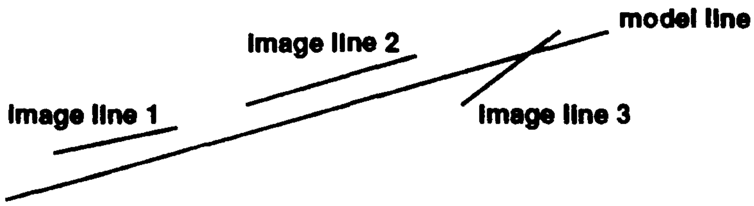

Figure 9.5. Many-to-one relationship of image lines to model lines.
Image lines 1, 2, and 3 may all be matched with the model line.

These five aspects of two-dimensional line matching could be combined to produce one value to describe the confidence for the match of one image line to one model line. Summing individual matches with highest confidences provides the most likely matching between image edges and model line segments, yet it remains to be determined if it is the correct matching. Beveridge [BEV90] includes an error term for model line segments that are omitted from the matching process. Many graph-based matching algorithms outlined by Ballard [BAL82] are NP-complete problems. Beveridge's method defaults to continually permuting various matching combinations in order to find the optimum matching.

C. MATCHING IMPLEMENTATION FOR VERTICAL LINE SEGMENTS

Assuming that the robot's camera is in a fixed position and that the robot is on a stable platform, the pose (or configuration) of the robot can be described in three degrees of freedom (x_0, y_0, θ_0). With this assumption, all vertical lines in the real world will be vertical in any image. Therefore, matching only vertical lines should provide a simple solution for the pose determination/correction problem. Since all line segments will be vertical, rotational differences between the image lines and model lines may be neglected, and the translational distance, d, can be simplified to be the horizontal angle from the midpoint of the image line to the model line.

Considering an image plane M that exists at focal length f from the center of the lens system of the robot's camera, the vertical edges extracted from the image will appear as in Figure 9.6. For a given estimated pose (x_0, y_0, θ_0), an overview of the robot and the image plane M (Figure 9.6) will be situated in the operating environment model as in Figure 9.7. The environment floor is the xy plane and heights of modelled features are expressed in the z-axis. Image plane M is parallel with the z-axis.

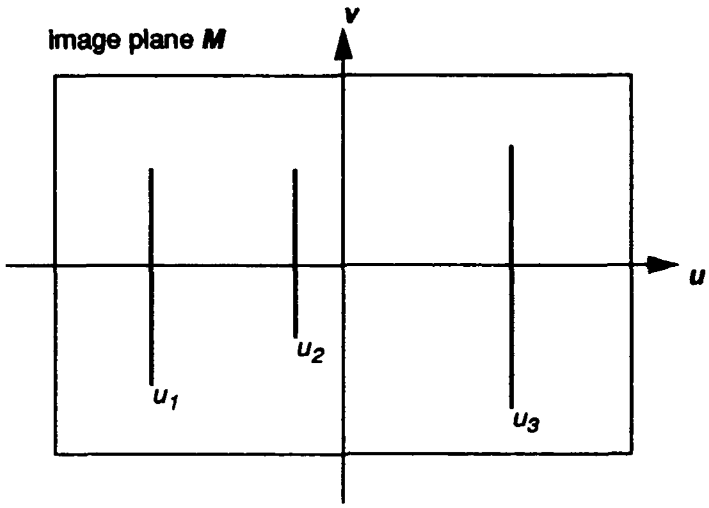

Figure 9.6. The image plane M. The image is centered on image axes u,v with three extracted vertical lines at u_1, u_2, and u_3.

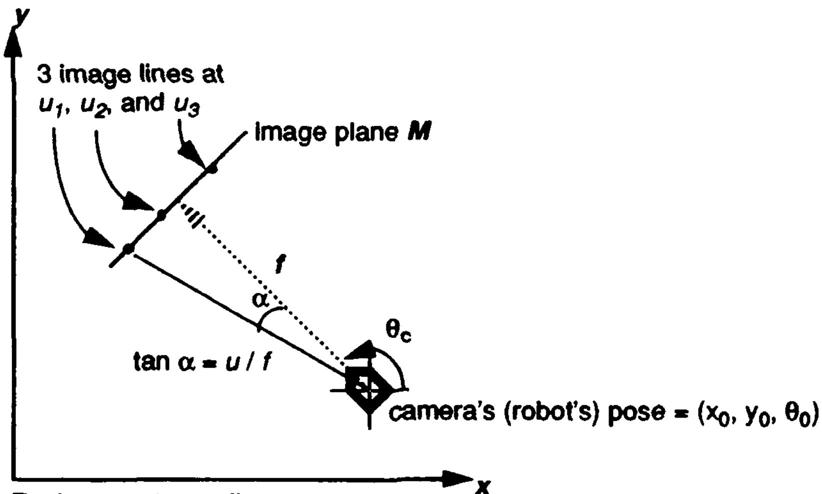

Figure 9.7. The robot in the environment coordinate system. The robot's pose is defined by (x_o, y_0, θ_0). The image plane M exists at the focal length (f) from the robot's camera. Vertical lines extracted from the input image in Figure 9.6 (at image u-coordinate positions u_1, u_2, and u_3) can be described by an angle α from the center of the image.

A two-dimensional view of the environment model is constructed given the three-dimensional model, an assumed pose, and the focal length f (Figure 9.8). The assumed pose is provided by the robot's dead reckoning capabilities from wheel motion and is used knowing that it is an approximation. Therefore, the matching problem to evaluate how well the image edges from the image plane M fit onto the model lines in the two-dimensional plane M'. For this implementation using only vertical lines, quantifying the match between an edge and a model line will be based only upon the translational distance d, if the endpoint inclusion requirement is met.

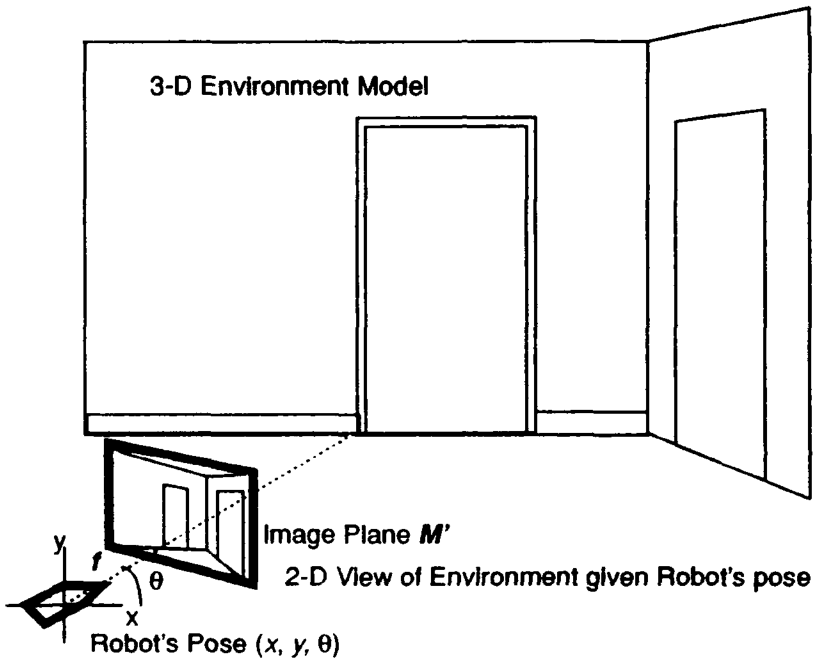

Figure 9.8. The image plane M'. M' is the two-dimensional view of the modelled environment for the given pose (x_0,y_0,θ_0) of the robot (camera). M' can be thought to exist at a distance equal to the focal length, f, of the camera's lens system.

The line matching provides connectivity between the extracted image lines and the vertical features of the environment model. Thus, the three most significant extracted image lines can be matched with three vertical model lines as in Figure 9.9. Vertical image lines u_1, u_2, and u_3, from the image plane M, are best matched with vertical model features L1, L2, and L3, respectively. This does not ensure that this is the correct matching, but it will provide a means to calculate a possible pose for the robot. This method also assumes that the three most significant vertical image lines are produced from modelled linear features.

Since backtracking may be necessary for recovering from successive attempts in the pose determination algorithm, every image edge in M must be able to support a linked list to all of the possible matches to the model line segments in M'. Appendix C contains the code for this implementation, titled *vertmatch*. A data structure named MATCHTYPE in the file *match_types.h* (Appendix A) is used for the association of an extracted edge with a model line segment. Each image edge data structure has a pointer to the head of the list of possible MATCHTYPEs and a separate pointer for the best. The different possible matches of an image edge to various model line segments are ordered by horizontal angular difference between the location of image edge and model line. The smallest difference, that being the best match, is at the head of the match list and all subsequent matches follow. The pose determination/correction algorithm in Chapter X utilizes only the three most significant extracted image edges (i.e. the three image lines with the greatest values for d_{major}). Thus the *vertmatch* implementation creates a list of MATHCTYPEs for the three most significant edges.

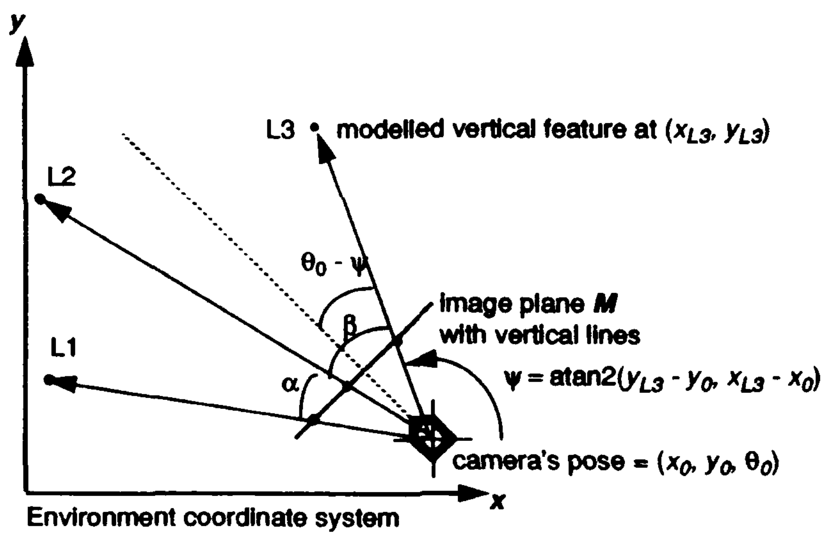

Figure 9.9. Modelled vertical features matched with extracted vertical edges from an image. L1, L2, and L3 are the labels assigned to the vertical lines matched to the three most significant vertical image lines in found in M. Angle ψ is the orientation of the line from the camera's position to the location of a vertical line. The modelled vertical line should then appear at an angle (θ_0 - ψ) from the center of plane M', the 2D view of the model environment (Figure 9.8).

X. CORRECTING ROBOT'S POSE

With the assumption that the robotic vehicle is a stable platform with its camera at a fixed position, there are only three degrees of freedom in the model space (x, y, and θ - corresponding to the variables of the robot's pose) that need to be determined. If considering only vertical lines for matching, it is possible to determine the camera's pose from only the three most significant vertical lines in an image, provided that they are all produced from features represented in the three-dimensional environment model.

A. POSE DETERMINATION

The only known, accurately measured quantities that are available for pose determination are the xy coordinate map locations of the three vertical model lines and the horizontal angular differences between the extracted vertical edges u_1, u_2, and u_3 from Figure 9.6. Let us denote two of these angles, the angle between the leftmost (u_1) and middle (u_2) image lines and the angle between the middle (u_2) and rightmost (u_2) image lines as α and β, respectively. If the best matches for u_1, u_2, and u_3 are used (vertical model lines L1, L2, and L3), angle α is the expected viewing angle between L1 and L2, and β is the expected viewing angle between L2 and L3. There is exactly one location that will have viewing angles α and β between model lines L1, L2, and L3. If the distance between two vertical model features is thought of as the side of a triangle (a) and the associated viewing angle (α) as the opposite angle of the triangle, an infinite number of possible locations for the viewing angle exist on a circle that circumscribes all possible triangles (Figure 10.1). For two triangles, one with side a and opposite angle α, and the other with side b and angle β, two circles of possible viewing locations can be constructed as in Figure 10.2. Sides a and b are therefore chords of the two circles. There are two intersections of the circles, provided that the circles are not identical. One intersection is at the location of vertical model line L2 and the other is the possible position of the camera (x_{poss}, y_{poss}). The camera must be at this position for this line matching, given the viewing angles α and β between the three major vertical image edges in M.

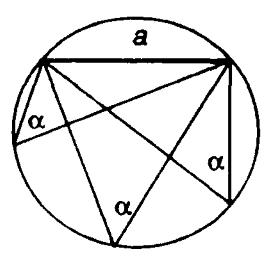

Figure 10.1. Triangles with side a and opposite angle α, circumscribed by a circle.

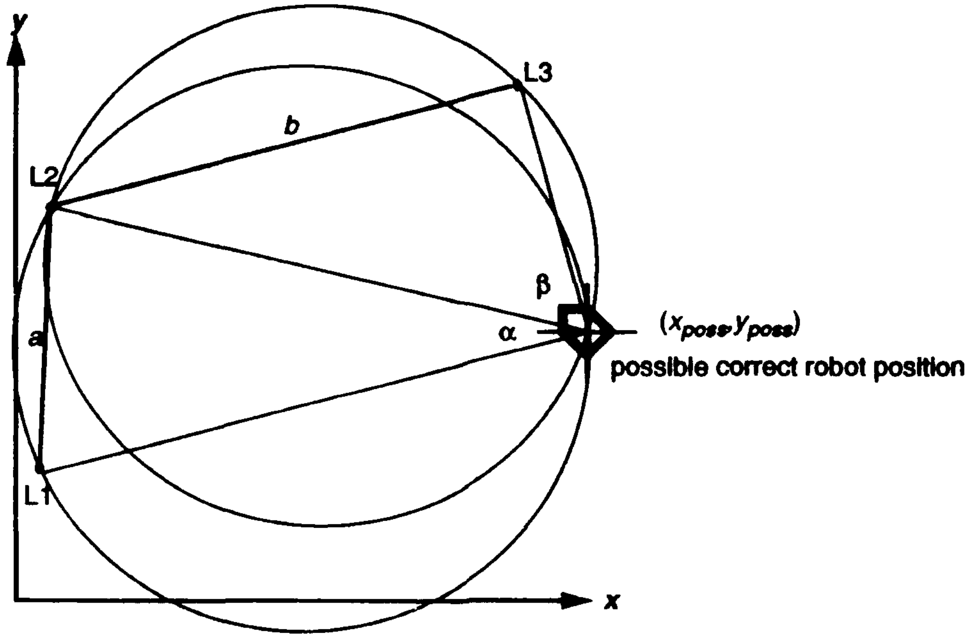

Figure 10.2. Geometry for pose determination. Given measured locations of vertical features in the environment model (L1, L2, and L3), circle chords a and b can be calculated. Angles α and β are measured from the image plane and are defined by the angular difference between the vertical extracted edge features from the image. With chords a and b and angles α and β, two circles can be constructed such that their intersections will exist at the position of model line L2 (x_{L2}, y_{L2}) and a possible robot position (x_{poss}, y_{poss}).

Once a position has been determined, calculation of the camera's orientation in the environment is simple. Since the locations of the vertical modelled features are known in the environment map (x_{Li}, y_{Li}), the angular orientation, ψ, from the possible position of the camera (x_{poss}, y_{poss}) is atan2$(x_{Li} - x_{poss}, y_{Li} - y_{poss})$. The angular location of these vertical model lines from the vertical centerline axis in the two-dimensional plane M' is $\theta_c - \psi$. Averaging the differences between angles of the model lines from the center of the plane M' $(\theta_c - \psi)$ and the angles of the image lines from the center of the image plane M provides a rotational correction. The correction is added to the orientation of the input (dead-reckoning) pose, θ_0, to get the possible pose orientation, θ_{poss}. The pose for the camera based upon the given matching is therefore determined $(x_{poss}, y_{poss}, \theta_{poss})$.

B. POSE VERIFICATION

The next best match combination will be performed by selecting the next match with the least horizontal angle difference between the vertical image edge and model line segment out of all three image line match lists. For this new combination of three matches, another possible pose is calculated by the above method. This is done for the n best match combinations. An exception to this rule is to skip matches where two or more image lines are matched to the model line segment. This method will provide, at most, n possible poses from the n match combinations.

Since a unique pose can be determined from any given matching combination for the three most significant image edges, the determination of the correct pose of the robot must rely upon a means to verify the pose and the matching combination. This topic is described by Heller and Stenstrom [HEL89]. For a given possible pose (x_{poss}, y_{poss}, θ_{poss}), locations for all the model line segments can be calculated if they were to be viewed in the image plane M. Comparing all of the extracted image lines to see how many fit the repositioned model lines provides a means to evaluate the possible pose. However, since many image lines are not produced from modelled features using the number of good image to model line fittings alone can not guarantee the correct pose.

If some credibility is expected of the estimated pose for which the two-dimensional view of the model (M')was constructed from, then calculated possible poses should be close to the estimated pose. Thus, an evaluation ($eval_{poss}$) combining the number of image edge to model line segment fittings (m_{poss}) with the distance between the possible pose and the estimated pose (d_{poss}) will be of the form

$$eval_{poss} = \frac{m_{poss}}{d_{poss}}. \qquad \text{(Eq 10.1)}$$

The possible pose with the greatest value for ($eval_{poss}$) will then be considered the corrected pose. Results of this method are shown in Chapter XI.

XI. PATTERN MATCHING AND POSE DETERMINATION RESULTS

The results from three trials are shown in this chapter. The actual camera pose is measured from a reference point in the operating environment. The estimated pose for developing the two-dimensional view of the model environment was encoded in the program *vertmatch* to simulate dead-reckoning errors. Each trial has a series of four figures. The first figure is the input image with the extracted vertical edges. The second figure is a wire frame two-dimensional view of the environment model based upon the estimated pose. The third figure is the wire-frame model superimposed over the input image. The fourth figure is a wire-frame model based upon the corrected pose superimposed over the image. The inputs to the *vertmatch* routine are the image, the environment model, and the estimated pose. Output from the *vertmatch* routine listing the matches and possible poses is shown between the third and fourth figures of each trial.

A. TRIAL 1

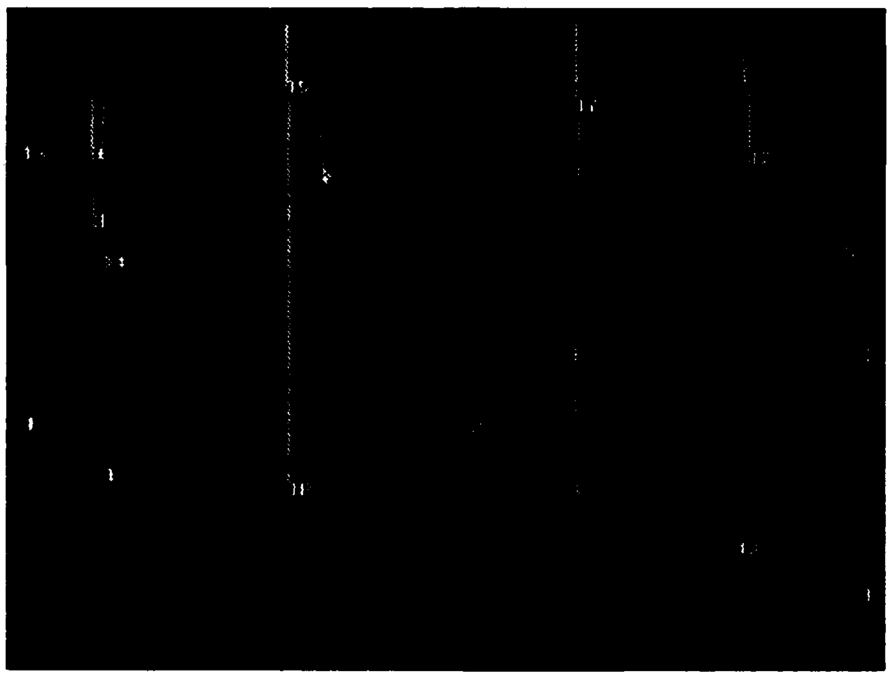

Figure 11.1. Trial 1, input image with extracted edges.
Actual camera pose: $x_0 = 60.0$ inches, $y_0 = 366.0$ inches, $\theta_0 = 253.0°$.

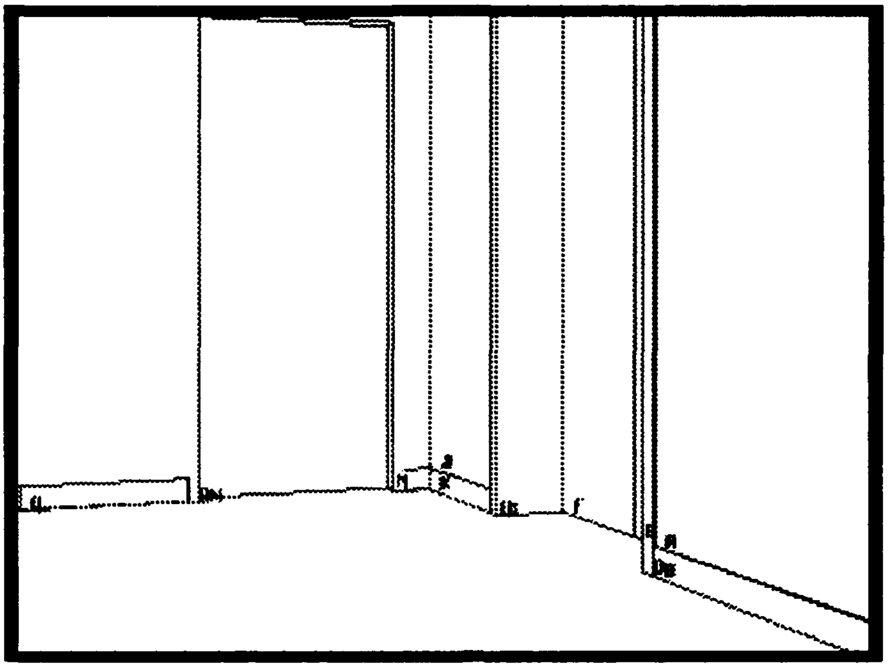

Figure 11.2. Trial 1, 2D wire-frame view of model based upon estimated pose. Estimated pose: $x_0 = 60.0$ inches, $y_0 = 366.0$ inches, $\theta_0 = 250.0°$.

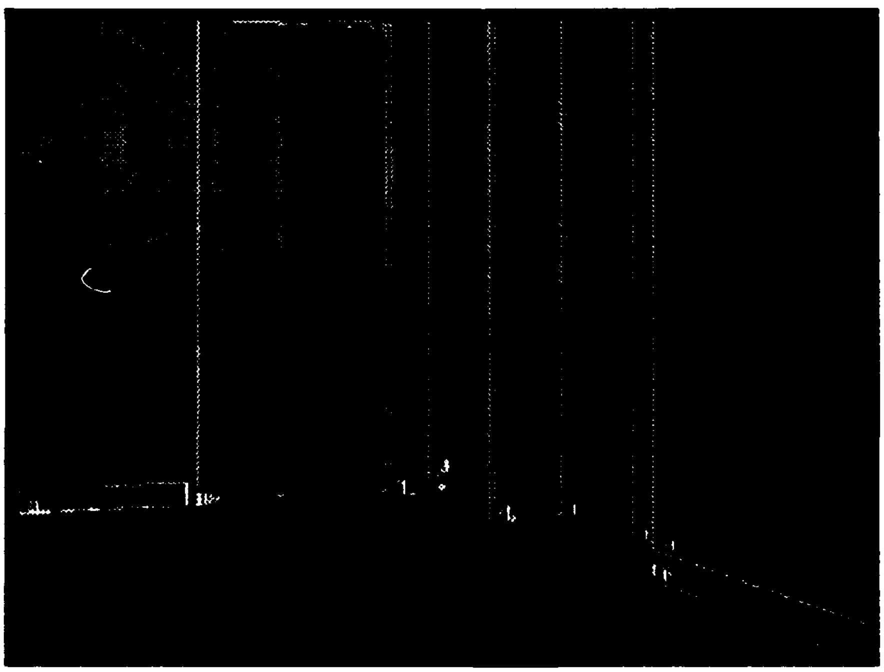

Figure 11.3. Trial 1, 2D wire-frame view superimposed over input image.

```
Script started on Thu Mar 19 12:12:18 1992
turing 1% vertmatch 060366253.pic
vertmatch> 060366253.pic xsize= 646 ysize= 486 pixels= 313956
 lines found in image written to: 'lines.text'
 Number lines found in 060366253.pic = 17

DR (input) pose: x = 60.00, y = 366.00, theta = 250.00(-1.92 rads)

Nr Vertical Model Lines = 20

Determine 3 major image lines for matching: -----
 1) line 13, angle from center = -0.1819
 2) line 10, angle from center = 0.0980
 3) line 8, angle from center = -0.0814

Image line matchlists to Model line NAME(angle difference): -----
left (edge 10) >
 N(0.0526) M(-0.0634) L(-0.0675) I(-0.0904) J(-0.0904) H(-0.1279)
 G(-0.1321) F(-0.1737) E(-0.2176) D(-0.2224) C(-0.2288) A(-0.2300)

middle (edge 8) >
 F(0.0057) E(-0.0382) D(-0.0430) G(0.0473) C(-0.0494) A(-0.0506)
 H(0.0515) J(0.0890) I(0.0890) L(0.1119) M(0.1160) N(0.2320)

right (edge 13) >
 A(0.0498) C(0.0511) D(0.0575) E(0.0623) F(0.1062) G(0.1478)
 H(0.1520) J(0.1895) N(0.3325)

Pose determination: -----
 [ N, F, A] x= 55.42 y= 513.24 T= 147.3124 R= -0.3730 nr verify matches= 5
 *** BEST POSE MODIFIED ***
 [ N, E, A] x= 276.04 y= 256.66 T= 242.1306 R= -0.6891 nr verify matches= 0
 [ N, D, A] NO position: chord length < MIN_CHORD_LENGTH
 [ N, G, A] x= 51.84 y= 381.92 T= 17.8955 R= 0.0048 nr verify matches= 6
 *** BEST POSE MODIFIED ***
 [ N, C, A] NO position: chord length < MIN_CHORD_LENGTH
 [ N, A, A] NO position: 2 IMG_LINEs matched to same model LINE
 [ N, A, C] NO position: chord length < MIN_CHORD_LENGTH
 [ N, H, C] x= 58.49 y= 368.04 T= 2.5417 R= 0.0464 nr verify matches= 7
 *** BEST POSE MODIFIED ***
 [ N, H, D] x= 58.48 y= 379.09 T= 13.1814 R= 0.0087 nr verify matches= 7
 [ N, H, E] x= 86.10 y= 459.31 T= 96.8872 R= -0.2843 nr verify matches= 6
 [ M, H, E] x= 307.54 y= 268.49 T= 266.0523 R= -1.1971 nr verify matches= 0
 [ L, H, E] x= 307.41 y= 268.42 T= 265.9568 R= -1.1694 nr verify matches= 0
 [ L, J, E] NO position: chord length < MIN_CHORD_LENGTH
 [ L, I, E] NO position: chord length < MIN_CHORD_LENGTH
 [ I, I, E] NO position: 2 IMG_LINEs matched to same model LINE
 [ J, I, E] NO position: chord length < MIN_CHORD_LENGTH
 [ J, I, F] NO position: chord length < MIN_CHORD_LENGTH
 [ J, L, F] NO position: chord length < MIN_CHORD_LENGTH
 [ J, M, F] NO position: chord length < MIN_CHORD_LENGTH
 [ H, M, F] x= 276.86 y= 247.81 T= 246.9754 R= 0.8255 nr verify matches= 2
 [ G, M, F] x= 270.77 y= 246.45 T= 242.3155 R= 0.8120 nr verify matches= 2
 [ G, M, G] NO position: 2 IMG_LINEs matched to same model LINE
 [ G, M, H] x= 307.93 y= 268.25 T= 266.5063 R= -1.4201 nr verify matches= 0
 [ F, M, H] x= 340.40 y= 274.86 T= 294.8414 R= -1.5850 nr verify matches= 0
```

```
[ F, M, J] NO position: chord length < MIN_CHORD_LENGTH
[ E, M, J] NO position: chord length < MIN_CHORD_LENGTH
[ D, M, J] NO position: chord length < MIN_CHORD_LENGTH
[ C, M, J] NO position: chord length < MIN_CHORD_LENGTH
[ A, M, J] NO position: chord length < MIN_CHORD_LENGTH
[ A, N, J] x= 337.19 y= 307.83 T= 283.2245 R= -0.5412 nr verify matches= 0
[ A, N, N] NO position: 2 IMG_LINEs matched to same model LINE

Corrected pose: x= 58.49, y= 368.04, theta= 252.66(-1.8735 rads)
turing 2% exit
turing 3%
script done on Thu Mar 19 12:13:35 1992
```

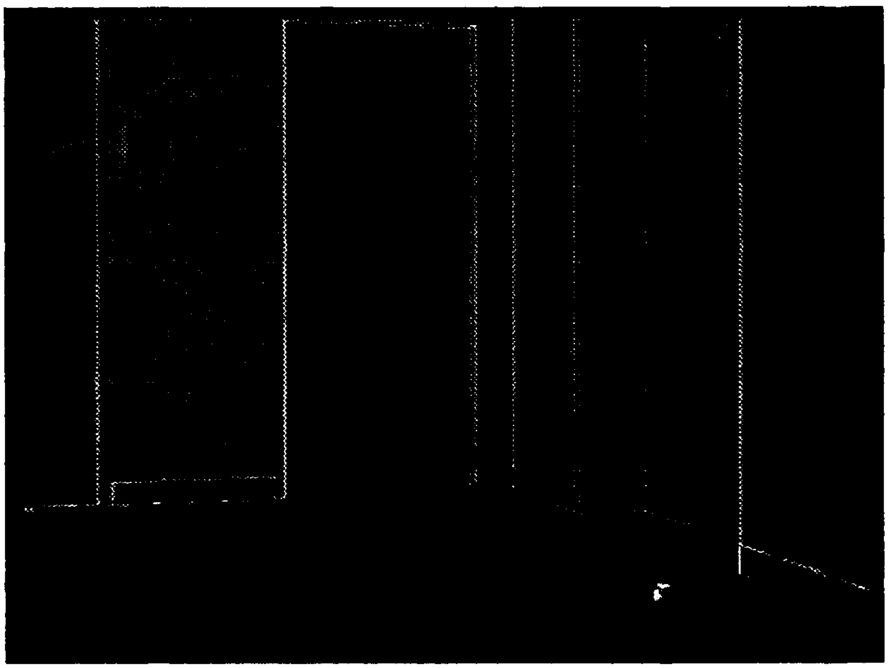

Figure 11.4. Trial 1, corrected 2D wire-frame view superimposed over input image.
Corrected pose: $x_0 = 58.49$ inches, $y_0 = 368.04$ inches, $\theta_0 = 252.66°$.
Translational error from actual pose = 2.54 inches, rotational error = 0.34°.

B. TRIAL 2

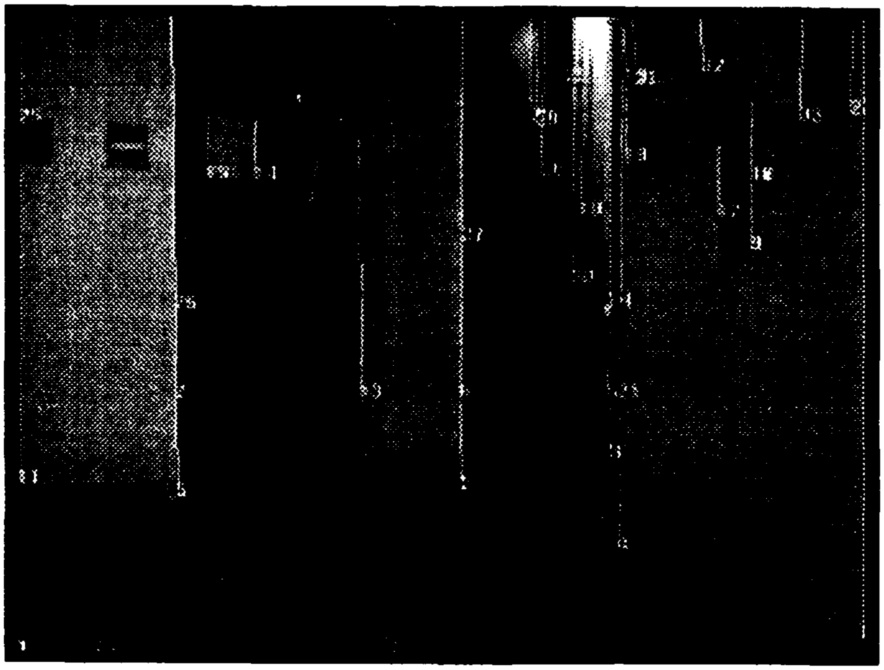

Figure 11.5. Trial 2, input image with extracted edges.
Actual camera pose: x_0 = 59.0 inches, y_0 = 366.0 inches, θ_0 = 249.0°.

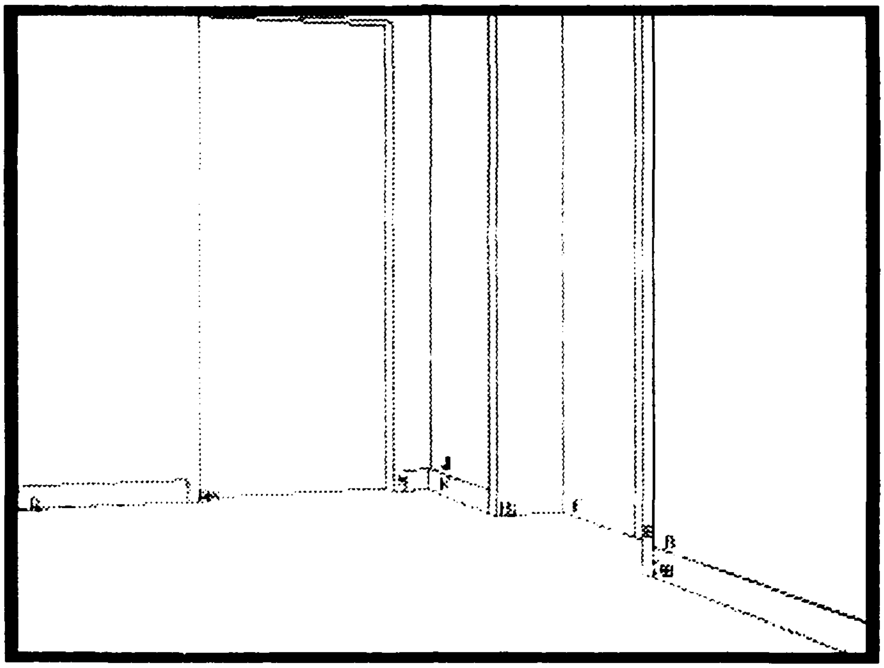

Figure 11.6. Trial 2, 2D wire-frame view of model based upon estimated pose.
Estimated pose: x_0 = 60.0 inches, y_0 = 366.0 inches, θ_0 = 250.0°.

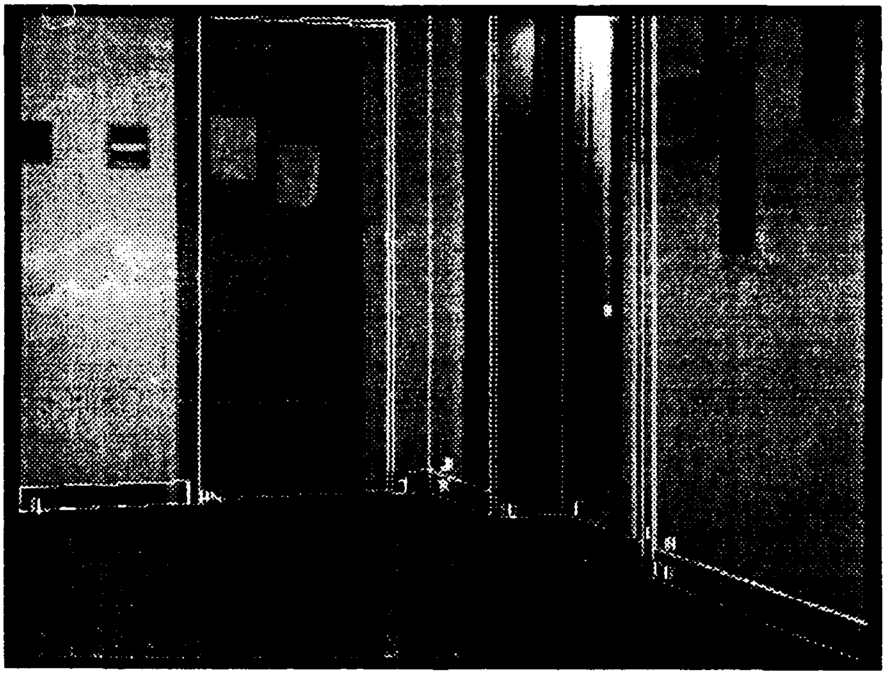

Figure 11.7. Trial 2, 2D wire-frame view superimposed over input image.

```
Script started on Thu Mar 19 12:19:42 1992
turing 1% vertmatch 059366249.pic
vertmatch> 059366249.pic xsize= 646 ysize= 486 pixels= 313956
 lines found in image written to: 'lines.text'
 Number lines found in 059366249.pic = 34

DR (input) pose: x = 60.00, y = 366.00, theta = 250.00(-1.92 rads)

Nr Vertical Model Lines = 20

Determine 3 major image lines for matching: -----
 1) line 21, angle from center = -0.1113
 2) line 26, angle from center = 0.1663
 3) line 13, angle from center = 0.0509

Image line matchlists to Model line NAME(angle difference): -----
left (edge 26) ->
 N(-[illegible]) M(-[illegible]) L(-0.1358) I(-0.1587) J(-0.1587) H(-0.1962)
 [illegible] F(-0.2859) D(-[illegible]) C(-0.2971) A(-0.2983)

middle (edge [illegible]) ->
 M([illegible]) [illegible] I(-0.0433) J(-0.0433) H(-0.0808) G(-0.0850)
 [illegible] E(-0.1704) D(-[illegible]) C(-0.1817) A(-0.1829)

right (edge [illegible])
 E(-[illegible]) [illegible]30) C(-0.0194) A(-0.0201) F(0.0357) G(0.0773)
 H([illegible]) [illegible] I(0.1190) L(0.1419) M(0.1460) N(0.2619)

[illegible] -----
 [illegible] 375.[illegible] T= [illegible].3823 R= -[illegible]44 nr verify matches= 10
 *** [illegible] ***
 [illegible] y= 370.16 T= [illegible].3981 R= -[illegible] nr verify matches= 17
 *** [illegible] ***
 [illegible] 366.45 T= [illegible] R= -[illegible] nr verify matches= 16
 *** [illegible]FIED ***
 [illegible] y= 377.14 T= 12.7495 R= -0.0496 nr verify matches= 15
 [illegible] 376.34 T= 12.3339 R= -0.0465 nr verify matches= 15
 [illegible] y= 446.84 T= 139.8[illegible] R= -[illegible]139 nr verify matches=

 [illegible] y= 458.[illegible] T= 191.6061 R= -0.7514 nr verify matches= 9
 [illegible] y= 458.37 T= 191.6061 R= -0.7514 nr verify matches= 9
 [illegible] y= 446.22 T= 206.9[illegible] R= -0.6209 nr verify matches= 8
 [illegible]tion: chord length < MIN_CHORD_LENGTH
 [illegible]tion: 2 IMG_LINEs matched to same model LINE
 [illegible]tion: chord length < MIN_CHORD_LENGTH
 [illegible]tion: 2 IMG_LINEs matched to same model LINE
 [illegible]tion: 2 IMG_LINEs matched to same model LINE
 [illegible]tion: 2 IMG_LINEs matched to same model LINE
 [illegible] y= 213.99 T= 252.6534 R= 1.0803 nr verify matches= 7
 [illegible] y= 259.29 T= 274.5149 R= -0.3348 nr verify matches= 0
 [illegible] y= 258.10 T= 272.0332 R= -0.3230 nr verify matches= 0
 [illegible]tion: 2 IMG_LINEs matched to same model LINE
 [illegible] y= 259.25 T= 272.3296 R= -0.2428 nr verify matches= 0
 [illegible] y= 257.00 T= 270.1404 R= -0.3378 nr verify matches= 0
 [illegible] y= 257.00 T= 270.1404 R= -0.3378 nr verify matches= 0
 [illegible] y= 257.15 T= 249.1[illegible]2 R= -0.4951 nr verify matches= 0
```

```
 J, D, M, x= [illegible].19 y= 267.95 T= 223.8134 R= -0.6992 nr verify matches= 0
 J, C, M, x= [illegible].59 y= 267.95 T= 221.4789 R= -0.7003 nr verify matches= 0
 J, A, M, x= [illegible].60 y= 267.68 T= 221.6025 R= -0.6970 nr verify matches= 0
 [illegible], A, M, x= [illegible].45 y= 267.44 T= 220.6821 R= -0.7016 nr verify matches= 0
 G, A, M, x= [illegible].34 y= 267.42 T= 220.5905 R= -0.7022 nr verify matches= 0
 F, A, M, x= [illegible].84 y= 266.64 T= 217.8188 R= -0.8255 nr verify matches= 0
 F, A, M, x= [illegible] y= 267.20 T= 216.0155 R= -0.6717 nr verify matches= 0
 E, A, M, x= [illegible].29 y= 266.64 T= 215.5532 R= [illegible].2840 nr verify matches= 0
 D, A, N, NO position: chord length < MIN_CHORD_LENGTH
 C, A, N, NO position: chord length < MIN_CHORD_LENGTH
 A, A, N, NO position: 2 IMG_LINEs matched to same model LINE

Corrected pose: x= [illegible], y= 366.45, theta= 249.02(-1.9370 rads)
turing [illegible]
turing [illegible]
script [illegible] Mar 19 12:20:56 1992
```

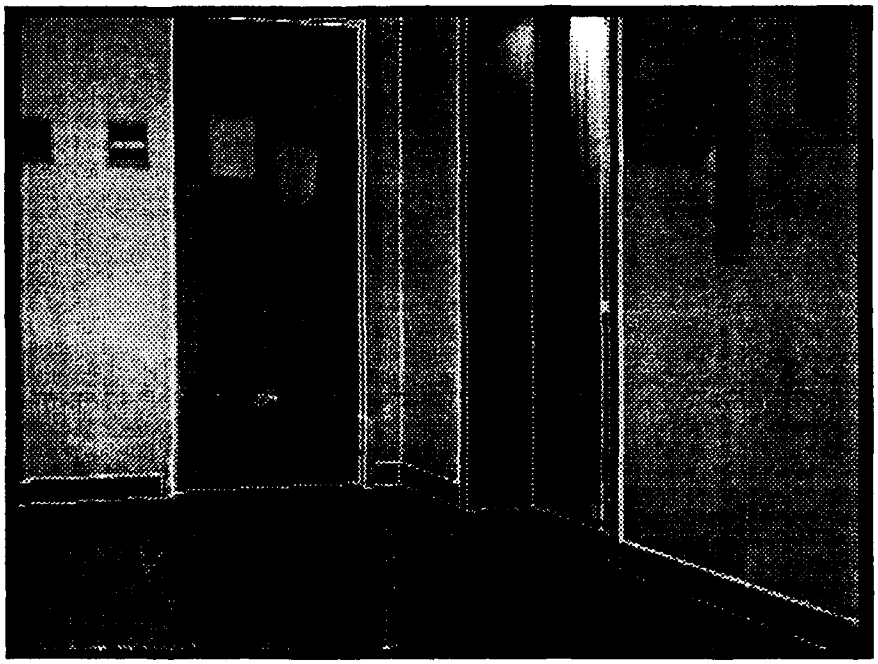

Figure 11.8. Trial 2, corrected 2D wire-frame view superimposed over input image.
Corrected pose: x_0 = 58.73 inches, y_0 = 366.45 inches, θ_0 = 249.02°.
Translational error from actual pose = 1.35 inches, rotational error = 0.02°.

C. TRIAL 3

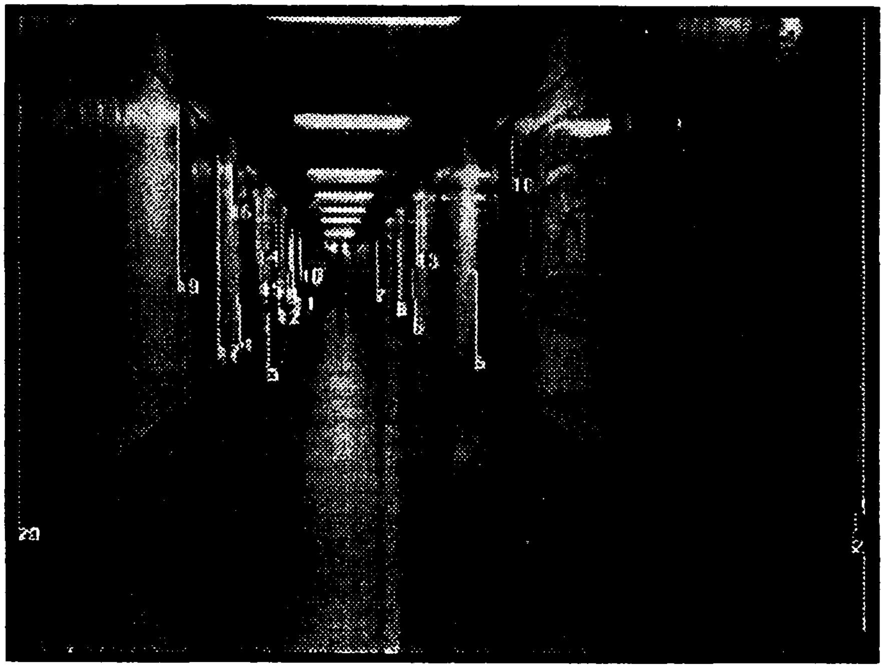

Figure 11.9. Trial 3, input image with extracted edges.
Actual camera pose: $x_0 = 43.0$ inches, $y_0 = 277.0$ inches, $\theta_0 = 356.0^{\circ}$.

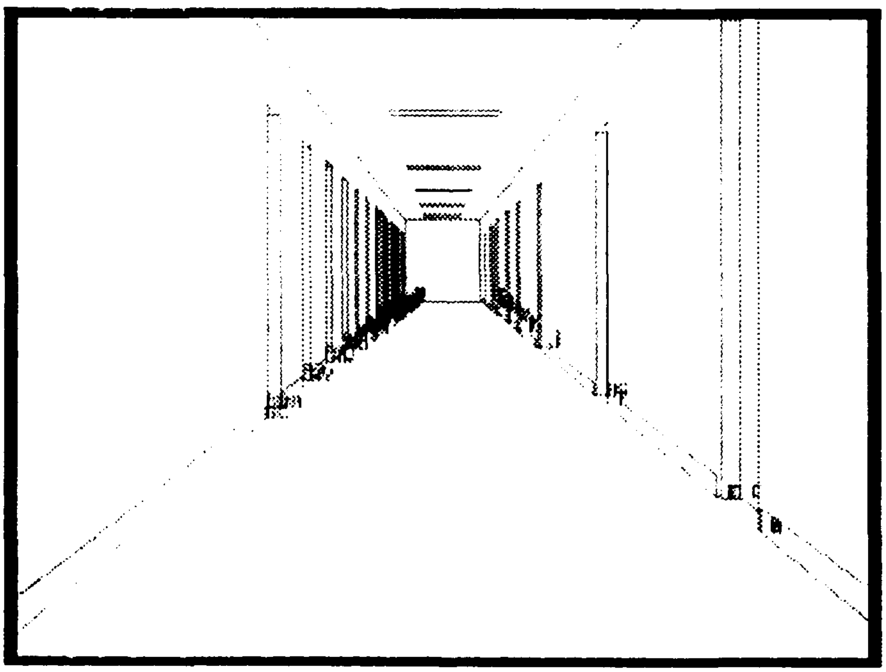

Figure 11.10. Trial 2. 2D wire-frame view of model based upon estimated pose.
Estimated pose: $x_0 = 48.0$ inches, $y_0 = 277.0$ inches, $\theta_0 = 000.0^{\circ}$.

Figure 11.11. Trial 3, 2D wire-frame view superimposed over input image.

```
Script started on Thu Mar 19 12:2[illegible]:08 1992
turing 1% vertmatch 043277356.pic
vertmatch [illegible]356.pic xsize= 646 ysize= 486 pixels= 313956
 lines found in image written to: 'lines.text'
 Number [illegible] found in 043277356.pic [illegible] 21

DR (input) [illegible]: x [illegible]48.00, y = 2[illegible].00, theta = 0.00(0.00 rads)

Nr Vertical [illegible] lines = 135

Determ[illegible] image lines for matching: -----
 1) line [illegible], angle from center = 0.1390
 2) line [illegible], angle from center = 0.1637
 3) line [illegible], angle from center = 0.1021

Image line match[illegible]s to Model line NAME(angle difference): -----
left (edge [illegible]) -
 BC(-[illegible]) [illegible](-[illegible].0550) BA(-0.0625) AZ(-0.0765) AY(-0.0818) AX(-0.0912)
 AW(-[illegible].094[illegible]) [illegible](-[illegible].2598) G(-0.2666) F(-0.2668) D(-0.3359) C(-0.3480)
 B(-0.[illegible])

middle (edge [illegible]) -
 BC(-0.02[illegible]) [illegible](-[illegible].0303) BA(-0.0378) A[illegible](-0.0518) AY(-0.0571) AX(-0.0665)
 AW(-0.[illegible]) [illegible](-[illegible]) G(-0.2418) F(-0.2421) D(-0.3112) C(-0.3232)
 B(-0.[illegible])

right (edge [illegible]) -
 BA(-[illegible]) [illegible](0.0066) BC(0.0074) AZ(-0.0[illegible]49) AY(-0.0202) AX(-0.0295)
 AW(-[illegible]) [illegible](-[illegible].0401) AU(-0.0428) AT(-[illegible]) AS(-0.0500) AR(-0.0540)
```

```
AQ(-0.0596) AP(-0.0605) AO(-0.0616) AN(-0.0625) AM(-0.0647) AL(-0.0654)
AK(-0.0663) N(-0.1422) M(-0.1442) L(-0.1490) K(-0.1506) J(-0.1614)
I(-0.1640) H(-0.1981) G(-0.2049) F(-0.2052) D(-0.2743) C(-0.2863)
B(-0.2981)

Pose determination: -----
[BC,BC,BA] NO position: 2 IMG_LINEs matched to same model LINE
[BC,BC,BB] NO position: 2 IMG_LINEs matched to same model LINE
[BC,BC,BC] NO position: 2 IMG_LINEs matched to same model LINE
[BC,BC,AZ] NO position: 2 IMG_LINEs matched to same model LINE
[BC,BC,AY] NO position: 2 IMG_LINEs matched to same model LINE
[BC,BC,AX] NO position: 2 IMG_LINEs matched to same model LINE
[BC,BB,AX] x= -2.43 y= 737.23 T= 462.9611 R= 0.9733 nr verify matches= 0
*** BEST POSE MODIFIED ***
[BC,BB,AW] x= -2.16 y= 734.34 T= 460.0836 R= 0.9748 nr verify matches= 0
[BC,BA,AW] x= 8.63 y= 619.54 T= 344.7975 R= -0.0743 nr verify matches= 12
*** BEST POSE MODIFIED ***
[BC,BA,AV] x= [illegible].98 y= 623.75 T= 349.0543 R= -0.0781 nr verify matches= 12
[BC,BA,AU] x= [illegible].91 y= 624.25 T= 349.5528 R= -0.0788 nr verify matches= 12
[BC,BA,AT] x= [illegible].98 y= 623.89 T= 349.1925 R= -0.0795 nr verify matches= 11
[BC,BA,AS] x= [illegible].04 y= 623.33 T= 348.6295 R= -0.0795 nr verify matches= 11
[BC,AZ,AS] x= [illegible] y= 298.72 T= 21.7457 R= -0.0466 nr verify matches= 13
*** BEST POSE MODIFIED ***
[BC,AZ,AR] x= [illegible].92 y= 354.76 T= 78.4074 R= -0.0593 nr verify matches= 15
[BB,AZ,AR] x= [illegible].28 y= 362.97 T= 86.6773 R= -0.0600 nr verify matches= 16
[BB,AZ,AQ] x= [illegible].80 y= 372.41 T= 96.3157 R= -0.0632 nr verify matches= 16
[BB,AY,AQ] x= [illegible].12 y= 172.41 T= 106.3377 R= -0.0423 nr verify matches= 15
[BB,AY,AP] x= [illegible].78 y= 240.96 T= 36.1426 R= -0.0580 nr verify matches= 14
[BB,AY,AO] x= [illegible].30 y= 252.74 T= 24.7655 R= -0.0606 nr verify matches= 14
[BB,AY,AN] x= [illegible] y= 260.39 T= 16.6734 R= -0.0625 nr verify matches= 15
*** BEST POSE MODIFIED ***
[BA,AY,AN] x= [illegible] y= 443.54 T= 16[illegible].6746 R= -0.0718 nr verify matches= 14
[BA,AY,AM] x= [illegible] y= 452.96 T= [illegible].1836 R= -0.0741 nr verify matches= 14
[BA,AY,AL] x= [illegible] y= 455.26 T= 179.5153 R= -0.0747 nr verify matches= 14
[BA,AY,AK] x= [illegible] y= 458.26 T= 182.5412 R= -0.0755 nr verify matches= 15
[BA,AX,AK] x= [illegible] y= 104.94 T= 173.6978 R= -0.0535 nr verify matches= 15
[BA,AW,AK] x= [illegible].35 y= -207.84 T= 491.2890 R= -0.0319 nr verifymatches=12
[AZ,AA,AK] x= [illegible] y= 380.50 T= 103.5455 R= -0.0639 nr verify matches= 14
[AY,AA,AK] x= [illegible] y= 560.79 T= 284.4956 R= -0.0726 nr verify matches= 15
[AX,AA,AK] x= [illegible] y= 842.19 T= 566.5463 R= -0.0781 nr verify matches= 9
[AW,AA,AK] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE
[AW,AA,[illegible]] NO position: 2 IMG_LINEs matched to same model LINE

Corrected p[illegible] x= [illegible]46.55, y= 260.39, theta= 356.42(-0.0625 rads)
turing 2% ex[illegible]
turing 3%
script done [illegible] Mar 19 12:22:18 199[illegible]
```

Figure 11.12. Trial 3, corrected 2D wire-frame view superimposed over input image.
Corrected pose: x_0 = 46.55 inches, y_0 = 260.39 inches, θ_0 = 356.42°.
Translational error from actual pose = 16.67 inches, rotational error = 0.42°.

XII. CONCLUSIONS

A. FEATURE EXTRACTION

The edge extraction implementation *fastedge*, provides linear features that, unlike the Hough transform, have good endpoint information. The implementation requires approximately 15 seconds to extract the linear edges from a 486 x 646 pixel image (313,956 pixels total) on the Personal Iris (35 MHz clock cycle). For a picture that has been "shrunken" to half of its width and height, the process requires only 3 seconds.

Selecting the appropriate constants C_1 through C_5 is important. The effects of modifying these parameters, outlined in Chapter VII, provide optimum values for use in a specific environment. The task of determining optimum values for a dynamic environment could be automated; however, it would require an extra scan through the image or else added hardware (e.g. a lightmeter).

The version implementing two static sets of angles used for the description of pixel gradient orientations could provide a very fast hardware implementation. Although it did not perform as well as averaging gradient orientations, the speedup advantages gained from implementing this method warrant more consideration and experimentation.

B. PATTERN MATCHING

Pattern matching is the "weak link" in the *vertmatch* implementation. Although the method is quick, it is crude. It was chosen as a means to provide a simple basis for the pose determination problem, therefore opting for the easier problem of matching only two-dimensional vertical line segments with the assumption that the three most significant edges extracted from the image are products of vertical modelled features. Many line matching implementations are graph based and therefore rely upon links between the extracted line segments that can only be inferred, but not accurately determined. Graph based matching algorithms are often NP complete thus leading to alternative methods like Beveridge's randomly permuting matches to find the optimum match.

C. POSE DETERMINATION

The pose determination algorithm was based upon simple vertical line matching in a known orthogonal environment and the geometry for its foundation allows a unique pose to be calculated simply and quickly. It was found to provide good estimates from three visual lines of bearings. Small errors between the actual and corrected poses in the three trials in Chapter XI could be from the algorithm, or from hardware floating-point

constraints involved with the trigonometric functions, or (most likely) from data measurement error from the tape measure.

The pose determination test trials were conducted at two different primary locations, viewing the elevator alcove and viewing the long axis of the hallway. Trials 1 and 2 in Chapter XI were conducted with the environment features of the elevator alcove approximately 20 feet from the camera which provided only 20 vertical model line segments. With relatively few model line segments for the three major image edges to be compared with, the matching was easy to check. Trial 3 involved much greater distances and over 100 vertical model line segments. The corrected pose could be more accurate if more iterations for possible matches and poses are conducted. However, the actual pose can be determined if all match combinations are tested exhaustively. Implementation of backtracking with some good heuristics about the matching process would be beneficial here. The long aspect of the hallway for pose determination is more difficult in terms of distances and possible matching combinations; however, other features (edges from baseboards and overhead lights) may be utilized to enhance the performance. Thus, a more general and more robust line matching implementation is required.

D. FOLLOW-ON WORK FOR YAMABICO-11

The methods developed in this thesis only have been directed towards the first objective of visual navigation in a known environment and has not dealt with the greater complexities of object recognition in an unknown environment. There remains much work to be done for continuing the implementation of visual navigation capabilities on Yamabico-11. Some of these areas are:

- Coding the methods in this thesis in 68020 microprocessor assembly language.
- Developing a vision based support programming language similar to the Mobile Motion Language.
- Implementing the environment model support routines in hardware.
- Developing a multiprocessor architecture to handle image edge extraction, two-dimensional environment view construction, line matching, and pose determination in parallel.
- Incorporating sensor fusion of sonar data with the vision navigation routines to check accuracy.
- Develop and implement fast, robust line matching method for all lines.
- Develop a method to positively determine if an edge is from a modelled linear feature.
- Develop object-based recognition methods to handle known and unknown objects.
- Develop an vision based algorithm that supports a reflexive behavior for obstacle avoidance.
- Develop methods to track moving obstacles.

APPENDIX A - DATA TYPES

A. FUNCTION: ATAN2

```
/*************************************************************************
/* file: atan2.c
/*
/* Provide a definition of the atan2() function.
/*
/*************************************************************************/

#define PI 3.141592653589793


double atan2(y,x)
  double x, y;
{
  if (x > 0.0) return (arctan(y/x));
  else if ((x < 0.0) && (y > 0.0)) return (arctan(y/x) + PI);
  else if ((x < 0.0) && (y < 0.0)) return (arctan(y/x) - PI);
  else if ((x < 0.0) && (y == 0.0)) return (PI);
  else if ((x == 0.0) && (y > 0.0)) return (PI/2.0);
  else if ((x == 0.0) && (y < 0.0)) return (-PI/2.0);
  else return (0.0);
}
```

B. DATA TYPE HEADER FILE: IMAGE_TYPES.H

```
/**********************************************************************
/* file: image_types.h
/*
/* This file holds the structure definitions for the different
/* types of in-memory images.
/* Types defined: CMAPIMAGE
/*                NPSIMAGE
/**********************************************************************/
#define RGBA 1 /* RGBA 24 bit images (alpha is 0xff filled) */
#define CMAPPED 2 /* color mapped images */
#define RGBAWITHALPHA 3 /* RGBA 32 bit images where alpha is read/saved
 in the image files. */

/* define a structure type for color mapped images */

struct cmapimage
{
  short *bitsptr; /* the bits for the short images */

   long nentries; /* the total number of entries in the color map */

   short *reds; /* ptr to the red entries of the color map */

   short *greens; /* ptr to the green entries in the color map */

   short *blues; /* ptr to the blue entries in the color map */

   long cmapoffset; /* color map offset, i.e. the first color we will us in the color map */
};
typedef struct cmapimage CMAPIMAGE; /* define a CMAPIMAGE type */


/* define a union so that the top level image structure's last pointer can point to several different
 kinds of images.*/

union imagedptr
{
   long *bitsptr; /* long images need no more data than a ptr to the bits. */
   CMAPIMAGE *cmapptr; /* a color mapped image must have the bits and a color map so
                           we need a complete structure. */
};


/* define the top level structure for the image */

struct image
{
   long type; /* image type */

   long xsize; /* xsize of the image */

   long ysize; /* ysize of the image */

   char *name; /* ptr to string naming the image */

   union imagedptr imgdata; /* ptrs to data for this type of image */

};
typedef struct image NPSIMAGE; /* define an NPSIMAGE type */
```

C. DATA TYPE HEADER FILE: MATCH_TYPES.H

```
/***************************************************************************
/* file: match_types.h
/*
/* This file holds the structure definitions for image edges and
/* match types to model lines.
/*
/* Types defined: EDGE
/*                POINT
/*                POSE
/*                MATCHTYPE
/*                IMG_LINE
/***************************************************************************/

typedef struct edge_region_type
{

   int active; /* boolean if past region is appended to a present region */

   long first_pixel; /* first and last pixels added to region */

   long last_pixel;

   long xmin; /* min & max pixels for previous row */

   long xmax;

   double avg_phi; /* for use with dynamic averaging of gradient orientation, phi */

   double sum_phi;

   /* Least Squares Fit mommments: */
   long m00; /* Number of pixels */

   double m10; /* Sum x */

   double m01; /* Sum y */

   double m11; /* Sum x*y */

   double m20; /* Sum x*x */

   double m02; /* Sum y*y */

   struct edge_region_type *next; /* ptr to the next EDGE */

} EDGE;



struct point_type
{

   double x,y; /* x,y coordinates of the pixel endpoints */

};
typedef struct point_type POINT;
```

```
struct pose_type
{
  float x, y, theta;

};
typedef struct pose_type POSE;


typedef struct match_type
{

  LINE *line; /* ptr to LINE type (from Jim Stein's "graphics.c") */

   double angle_view_diff; /* angular difference between image and model lines in the image */

   float conf; /* confidence value for the match */

   float dist; /* distance between the *match LINE and IMG_LINE */

   float scale; /* ratio IMG_LINE->dmajor / MODEL_LINE->length */

   struct match_type *next;

} MATCHTYPE;


typedef struct line_type
{

   char name[3];

   POINT p1, p2; /* the 2 endpoints for the line */

   struct line_type *next; /* ptr to the next IMG_LINE in the image */

   /* Least Squares Fit mommments: ------------------------------ */
   long m00; /* Number of pixels */

   double m10; /* Sum x */

   double m01; /* Sum y */

   double m11; /* Sum x*y */

   double m20; /* Sum x*x */

   double m02; /* Sum y*y */

   double phi; /* Calculated normal orientation of IMG_LINE */

   double dmajor; /* Length of major axis of equivallent ellipse */

   double dminor; /* Length of minor axis of equivallent ellipse */

   double rho; /* Ratio dminor/dmajor */
```

```
/* Pattern Matching Information: -------------------------------- */
double angle_to_image_center;

MATCHTYPE *matchlist; /* List of matches to LINE types */

MATCHTYPE *pm; /* present match being considered */

} IMG_LINE;
```

APPENDIX B - EDGE EXTRACTION ROUTINES

A. IMPLEMENTATION: FINDEDGE.C

```
/*********************************************************************/
/* FILENAME: findedge.c
/* AUTHOR: Kevin Peterson
/* DATE: 05 December 1991
/*
/*DESCRIPTION: An image gradient program incorporating edge-finding.
/*      Displays edge-gradient image and associated lines.
/*
/* This application is designed for use on a Silicon-
/* Graphics Iris (r) workstation utilizing a .sgi
/* or similar rgb formatted image.
/* RGB values are of type LONG in the form AABBGGRR where:
/* AA - alpha value, 0-255
/* BB - blue component, 0-255
/* GG - green component, 0-255
/* RR - red component, 0-255
/*
/* SiliconGraphics graphics library functions used within
/* the display_bw_and_gradient_images() routine:
/* qdevice(), winset(), c3f(), move2(), draw2(),
/* swapbuffers(), reshapeviewport(), winclose(),
/* and lrectwrite().
/*
/* NPSIMAGE function routines borrowed courtesy of M.Zyda:
/* read_sgi_rgbimage(), get_empty_rgba_npsimage(),
/* get_empty_rgb_npsimage(), and rgbalong_to_bwlong().
/*
/*      Least Squares Fit method for line-finding from
/*      "Sonar Data Interpretaion for Autonomous Mobile Robots"
/*      by Y.Kanayama, T.Noguchi, & B.Hartman, 1990.
/*********************************************************************/
#include <gl.h> /* SiliconGraphics (r) graphic library */
#include <gl/image.h> /* SGI image structure library */
#include <device.h> /* Machine-dependent device library for keys and mouse-buttons */
#include <stdio.h> /* C standard i/o library */
#include <math.h> /* C math library for atan2() */

#include "image_types.h" /* Type definitions for NPSIMAGE, etc. */
#include "edge_types.h" /* Type definitions for EDGE, LINE, etc */

#include "npsimagesupport.h" /* Some NPSIMAGE functions */
#include "edgesupport.h" /* EDGE and IMG_LINE building functions */
#include "displaysupport.h" /* Graphics display functions */

#define THRESHOLD 5000000.0 /* for gradient magnitude threshold test */
```

```
main(argc, argv)
        int argc;
        char *argv[];
{
        NPSIMAGE *img1, /* input file_name.rgb col`  `mage */
                        *img2, /* black&white image */
                        *img3; /* gradient image */

        /* pointers to RGBA longs ("bitsptr"s) of respective NPSIMAGEs */
        long *ptr1, *ptr2, *ptr3;

        double dx, dy, Th = THRESHOLD*THRESHOLD;
        register int i = 0, /* counter for pixels in input image */
                    z = 0; /* counter for pixels in gradient image */
        EDGE *reg;

if(argc != 2) fatal("usage: findedge filename\n");

/* Read in input rgb image */
img1 = read_sgi_rgbimage(argv[1]);
if(img1 ==(NPSIMAGE *) NULL)
{
            fatal("File %s is a NULL image.\n",img1->name);
}
Xdim = img1->xsize; /* else set global Xdim and ptr1 */
ptr1 = img1->imgdata.bitsptr;
printf("findedge:> %s xsize= %d ysize= %d pixels= %d\n",
        img1->name,img1->xsize,img1->ysize, (img1->xsize*img1->ysize));


/* Declare new NPSIMAGEs */
if((img1->type == RGBAWITHALPHA) || (img1->type == RGBA))
{
        img2 = get_empty_rgba_npsimage(Xdim,img1->ysize,img1->name);
        img3 = get_empty_rgba_npsimage((Xdim-2),img1->ysize-2,"findedge");
}
else
{
        fatal("Unknown or c-mapped image type: %d.\n",img1->type);
}
ptr2 = img2->imgdata.bitsptr;
ptr3 = img3->imgdata.bitsptr;

/* The scan of an RGB image is from the bottom row -> up,
traversing the rows left to right. */

/* In order for the Sobel operator to be calculated for a specific pixel,
all eight surrounding pixels must have an absolute (black & white)
light intensity calculated. Function rgbalong_to_bwlong() performs
this task. */

/* Due to the nature of the Sobel operator, the pixels in the top and
bottom rows as well as pixels in the leftmost and rightmost columns
of the input image will not be calculated. In order to start cal-
culating the Sobel operators, the first 2 rows of the input image
must be converted to black & white light intensity values. */
```

```
/* Calculate bw values for first 2 rows of input image. */
for(i=0; i<(2*Xdim)+2; ++i)
{
        rgbalong_to_bwlong(ptr1[i],&ptr2[i]);
}

for(i = Xdim + 1; i < (Xdim*(img1->ysize-1))-1; ++i)
{
/* Convert color(img1) to b/w(img2) for pixel on next row up and one
        pixel over to the right so that all eight neighbors of pixel i
        have black & white light intensities. */

        rgbalong_to_bwlong(ptr1[i+Xdim+1],&ptr2[i+Xdim+1]);
        /* Ensure pixel i is not in leftmost ro rightmost column */
        if((i%Xdim != 0) && (i%Xdim != Xdim-1))
            {
            /* Calculate dx,dy via Sobel operator for pixel i. */

            dx = (-ptr2[i+Xdim-1] + ptr2[i+Xdim+1]
                 -(2 * ptr2[i-1])+ (2 * ptr2[i+1])
                 -ptr2[i-Xdim-1] + ptr2[i-Xdim+1]);

            dy = ( ptr2[i+Xdim-1] + (2*ptr2[i+Xdim]) + ptr2[i+Xdim+1]
                 -ptr2[i-Xdim-1] - (2*ptr2[i-Xdim]) - ptr2[i-Xdim+1]);

            if((dx*dx)+(dy*dy) > Th)
            {
                pixel_membership(z,atan2(dy,dx));
                set_pixel_black(&ptr3[z]);
            }
            else
            {
                set_pixel_white(&ptr3[z]);
            }

            ++z; /* Increment the pixel counter z for the gradient image. */
            }

        /* If pixel i is in the leftmost column, do check_active_edges(). */
        else if(i%Xdim == 0)
            {
                check_active_edges();
            }
} /* endfor i */

/* Check remaining EDGEs for lines: */
reg = Past_edge_list_head;
while(reg != NULL)
{
        line_test(reg);
        reg = reg->next;
}

/* Write the lines list to file "lines.text". */
write_all_lines(argv[1],img3->xsize,img3->ysize);
printf("Number lines found in %s = %d\n", argv[1],Linecount);
/* Display the black&white and gradient images on the screen. */
display_bw_and_gradient_images(img2,img3,Line_list_head);

printf("findedge %s...done.\n",argv[1]);
}
```

B. IMPLEMENTATION: FASTEDGE.C

```
/*********************************************************************/
/* FILENAME: fastedge.c
/* AUTHOR: Kevin Peterson
/* DATE: 06 January 1992
/* DESCRIPTION: An image gradient program incorporating edge-finding.
/* Displays edge-gradient image and associated lines.
/*
/* This application is designed for use on a Silicon-
/* Graphics Iris (r) workstation utilizing a .sgi
/* or similar rgb formatted image.
/* RGB values are of type LONG in the form AABBGGRR where:
/* AA - alpha value, 0-255
/* BB - blue component, 0-255
/* GG - green component, 0-255
/* RR - red component, 0-255
/* SiliconGraphics graphics library functions used within the display_bw_and_gradient_images()
/* routine: qdevice(), winset(), c3f(), move2(), draw2(), swapbuffers(), reshapeviewport(), winclose(),
/* and lrectwrite().
/* NPSIMAGE function routines borrowed courtesy of M.Zyda: read_sgi_rgbimage(),
/* get_empty_rgba_npsimage(), get_empty_rgb_npsimage(), and rgbalong_to_bwlong().
/* Least Squares Fit method for line-finding from "Sonar Data Interpretaion for Autonomous Mobile
/* Robots" by Y.Kanayama, T.Noguchi, & B.Hartman, 1990.
/*********************************************************************/
#include <gl.h> /* SiliconGraphics (r) graphic library */
#include <gl/image.h> /* SGI image structure library */
#include <device.h> /* Machine-dependent device library for keys and mouse-buttons */
#include <stdio.h> /* C standard i/o library */
#include <math.h> /* C math library for atan2() */
#include "image_types.h" /* Type definitions for NPSIMAGE, etc. */
#include "edge_types.h" /* Type definitions for EDGE, LINE, etc */
#include "npsimagesupport.h" /* Some NPSIMAGE functions */
#include "edgesupport.h" /* EDGE and IMG_LINE building functions */
#include "displaysupport.h" /* Graphics display functions */

main(argc, argv)
 int argc;
 char *argv[];
{
 NPSIMAGE *img; /* input file_name.rgb color image */
 long *ptr; /* pointer to bitsptr of NPSIMAGE */
 long grmask = 0x0000ff00;
 double dx, dy, Th = THRESHOLD*THRESHOLD;
 register int i = 0, /* counter for pixels in input image */
              z = 0; /* counter for pixels in gradient image */
 EDGE *reg;

 if(argc != 2) fatal("usage: fastedge filename\n");

 /* Read in input rgb image */
 img = read_sgi_rgbimage(argv[1]);
 if(img ==(NPSIMAGE *) NULL) fatal("File %s is a NULL image.\n",img->name);
 printf("fastedge > %s xsize= %d ysize= %d pixels= %d\n",
    img->name,img->xsize,img->ysize, (img->xsize*img->ysize));

 Line_list_head = fastlines(img);

 display_line_image(img,Line_list_head);
printf("fastedge %s...done.\n",argv[1]);
}
```

C. IMPLEMENTATION: VERTEDGE.C

```
/*********************************************************************/
/* FILENAME: vertedge.c
/* AUTHOR: Kevin Peterson
/* DATE: 30 January 1992
/*
/* DESCRIPTION: Same a fastedge.c but only returns vertical edges from image.
/*
/*********************************************************************/

#include <gl.h> /* SiliconGraphics (r) graphic library */
#include <gl/image.h> /* SGI image structure library */
#include <device.h> /* Machine-dependent device library */
            /* for keys and mouse-buttons */
#include <stdio.h> /* C standard i/o library */
#include <math.h> /* C math library for atan2() */

#include "image_types.h" /* Type definitions for NPSIMAGE, etc. */
#include "edge_types.h" /* Type definitions for EDGE, LINE, etc */

#include "npsimagesupport.h" /* Some NPSIMAGE functions */
#include "edgesupport.h" /* EDGE and IMG_LINE building functions */
#include "vertsupport.h" /* Vertical EDGE and IMG_LINE supplement */
#include "displaysupport.h" /* Graphics display functions */


main(argc, argv)
 int argc;
 char *argv[];
{
 NPSIMAGE *img; /* input file_name.rgb color image */

 if(argc != 2) fatal("usage: vertedge filename\n");

 /* Read in input rgb image */
 img = read_sgi_rgbimage(argv[1]);
 if(img ==(NPSIMAGE *) NULL) fatal("File %s is a NULL image.\n",img->name);
 printf("vertedge> %s xsize= %d ysize= %d pixels= %d\n",
    img->name,img->xsize,img->ysize, (img->xsize*img->ysize));

 Line_list_head = vertlines(img);

 display_line_image(img,Line_list_head);

 printf("vertedge %s...done.\n",argv[1]);
}
```

D. FILE: NPSIMAGESUPPORT.H

```
/***********************************************************************/
/* FILENAME: npsimagesupport.h
/* AUTHOR: Kevin Peterson
/* DATE: 29 January 1992
/*
/*DESCRIPTION: Collection of basic npsimage functions.
/*
/*      NPSIMAGE *get_empty_rgb_npsimage (long xsize, long ysize, char name[])
/*      NPSIMAGE *get_empty_rgba_npsimage (long xsize, long ysize, char name[])
/*      NPSIMAGE *read_sgi_rgbimage (char filename[])
/*      void write_sgi_rgbimage (char filename[], NPSIMAGE *img)
/*
/***********************************************************************/

/*-----------------------------------------------------------------*/
/* NPSIMAGE *get_empty_rgb_npsimage (long xsize, long ysize, char name[])
/*
/* The following function reads in an SGI RGB image as an NPSIMAGE.
/* - coutesy of M. Zyda, Naval Postgraduate School
/*-----------------------------------------------------------------*/

NPSIMAGE *get_empty_rgb_npsimage(xsize,ysize,name)

  long xsize, ysize; /* the size the image should be in pixels */

  char name[]; /* name to attach to the image */

{

NPSIMAGE *img; /* ptr to an NPSIMAGE */


/* allocate an NPSIMAGE header */
img = (NPSIMAGE *)malloc(sizeof(NPSIMAGE));

/* set the type of NPSIMAGE to RGBA */
img->type = RGBA;

/* record the widths and height */
img->xsize = xsize;
img->ysize = ysize;

/* allocate space for the name of the image */
img->name = (char *)malloc(strlen(name)+1);

/* copy the name into the string allocated */
strcpy(img->name,name);

/* allocate the memory for the bitmap of the image */
img->imgdata.bitsptr =
(long *)malloc(sizeof(long) * img->xsize * img->ysize);

/* return a ptr to this empty image */
return(img);

}
```

```
/*------------ --------------------------------------------------------*/
/* NPSIMAGE *get_empty_rgba_npsimage (long xsize, long ysize, char name[])
/*
/* The following function reads in an SGI RGB image as an NPSIMAGE.
/* - coutesy of M. Zyda, Naval Postgraduate School
/*--------------------------------------------------------------------*/

NPSIMAGE *get_empty_rgba_npsimage(xsize,ysize,name)

  long xsize, ysize; /* the size the image should be in pixels */

  char name[]; /* name to attach to the image */

{

NPSIMAGE *img; /* ptr to an NPSIMAGE */


/* allocate an NPSIMAGE header */
img = (NPSIMAGE *)malloc(sizeof(NPSIMAGE));

/* set the type of NPSIMAGE to RGBA */
img->type = RGBAWITHALPHA;

/* record the widths and height */
img->xsize = xsize;
img->ysize = ysize;

/* allocate space for the name of the image */
img->name = (char *)malloc(strlen(name)+1);

/* copy the name into the string allocated */
strcpy(img->name,name);

/* allocate the memory for the bitmap of the image */
img->imgdata.bitsptr = (long *)malloc(sizeof(long) * img->xsize * img->ysize);

/* return a ptr to this empty image */
return(img);

}
```

```
/*---------------------------------------------------------------*/
/* NPSIMAGE *read_sgi_rgbimage (char filename[])
/*
/* The following function reads in an SGI RGB image as an NPSIMAGE.
/* - coutesy of M. Zyda, Naval Postgraduate School
/*---------------------------------------------------------------*/


NPSIMAGE *read_sgi_rgbimage(filename)

  char filename[]; /* input filename */

{

 register IMAGE *image;

 NPSIMAGE *img; /* ptr to an NPSIMAGE structure */

 register int x,y; /* temp indices for each line of
 data from the sgi image. */

 long *ptr; /* temp pointer for each word of the
 NPSIMAGE long image. */

 short rbuf[4096], /* temp arrays to hold scratch info for */
 gbuf[4096], /* processing an sgi image */
 bbuf[4096],
 abuf[4096];



 /* open an sgi image */
 if( (image=iopen(filename,"r")) == NULL )
 {
   fprintf(stderr,"read_sgi_rgbimage: can't open input file %s\n",filename);
   return((NPSIMAGE *)NULL);
 }

 if(image->zsize<3)
 {
  fprintf(stderr,"read_sgi_rgbimage: this is not an RGB image file\n");
   return((NPSIMAGE *)NULL);
 }

 /* here we should allocate an NPSIMAGE */
 if(image->zsize == 3)
 {
   /* just allocate an rgb image */
   img = get_empty_rgb_npsimage(image->xsize,image->ysize,filename);
 }
 else
 {
   /* get an image with alpha */
   img = get_empty_rgba_npsimage(image->xsize,image->ysize,filename);
 }

 /* get a pointer to the NPSIMAGE longs */
 ptr = img->imgdata.bitsptr;
```

```
/* for each row of the image ... */
for(y=0; y < img->ysize; y=y+1)
{
  /* read a row of reds */
  getrow(image,rbuf,y,0);

  /* read a row of greens */
  getrow(image,gbuf,y,1);

  /* read a row of blues */
  getrow(image,bbuf,y,2);

  /* if we have an image with alpha, get it */
  if(img->type == RGBAWITHALPHA)
  {
      getrow(image,abuf,y,3);
  }

  /* we must now step across the row and set each long integer
  of the NPSIMAGE format by combining the info from the sgi
  rows.
  */
  for(x=0; x < img->xsize; x=x+1)
  {

      /* compute the RGBa long to plug in and plug it in */
      if(img->type == RGBAWITHALPHA)
      {
          *ptr = rbuf[x] + (gbuf[x] << 8) + (bbuf[x] << 16) + (abuf[x] << 24);
      }
      else
      {
          /* RGBA image with alpha forced to 0xff */
          *ptr = rbuf[x] + (gbuf[x] << 8) + (bbuf[x] << 16) + (0xff << 24);
      }

  /* step the ptr to the next long */
  ptr++;

} /* end for */

}

/* we have set all the bytes of the image.
Now return the ptr to the NPSIMAGE structure.
*/
return(img);
}
```

```
/*------------------------------------------------------------------*/
/* void write_sgi_rgbimage (char filename[], NPSIMAGE *img)
/*
/* The following function wites an NPSIMAGE to a file.
/* - coutesy of M. Zyda, Naval Postgraduate School
/*------------------------------------------------------------------*/

void write_sgi_rgbimage(filename, img)

   char filename[]; /* output filename */

   PSIMAGE *img; /* ptr to an NPSIMAGE structure */

{
register IMAGE *image; /* a ptr to an sgi image structure */

 register int x,y; /* temp indices for each line of data from the sgi image. */

 long *ptr; /* temp pointer for each word of the NPSIMAGE long image. */

 long dimen; /* dimension of this image */

 short rbuf[4096], /* temp arrays to hold scratch info for */
 gbuf[4096], /* processing an sgi image */
 bbuf[4096],
 abuf[4096];


 /* set the dimension and zsize of this image */
 if(img->type == RGBAWITHALPHA)
 {
   dimen = 4;
 }
 else
 {
   /* RGBA image without alpha */
   dimen = 3;
 }

 /* open an sgi rgb image for writing */
 image=iopen(filename,"w",RLE(1),dimen,img->xsize,img->ysize,dimen);

 /* get a pointer to the NPSIMAGE longs */
 ptr = img->imgdata.bitsptr;

 /* for each row of the image ... */
 for(y=0; y < img->ysize; y=y+1)
 {

   /* we must now step across the row and decode each long integer
   of the NPSIMAGE format into the 16 bit shorts sgi requires */

   for(x=0; x < img->xsize; x=x+1)
   {
       /* get the colors from the longs */
       rbuf[x] = *ptr & 0x000000ff;
       gbuf[x] = (*ptr & 0x0000ff00) >> 8;
       bbuf[x] = (*ptr & 0x00ff0000) >> 16;
       abuf[x] = (*ptr & 0xff000000) >> 24;
```

```
        /* step the ptr to the next long */
        ptr++;
    }

    /* write a row of reds */
    putrow(image,rbuf,y,0);

    /* write a row of greens */
    putrow(image,gbuf,y,1);

    /* write a row of blues */
    putrow(image,bbuf,y,2);

    /* write a row of alphas, if any */
    if(img->type == RGBAWITHALPHA)
    {
        putrow(image,abuf,y,3);
    }

    }

/* we must close the output sgi image file */
iclose(image);

}
```

E. FILE: EDGESUPPORT.H

```
/***********************************************************************/
/* FILENAME: edgesupport.h
/* AUTHOR: Kevin Peterson
/* DATE: 29 January 1992
/*
/*DESCRIPTION: Collection of edge finding functions.
/*
/* void fatal(char message)
/* int gradient_angles_close(double r,double s)
/* int close_to_negative_pi(double phi)
/* int horizontal(EDGE *r)
/* EDGE *create_edge(long z,double phi)
/* void add_pixel_to_edge (long z, double phi, EDGE *r)
/* EDGE *combine_edges(EDGE *r1, EDGE *r2)
/* IMG_LINE *create_line (EDGE *r, double M20, double M11, double M02,
/* double Dmajor, double Dminor, double Rho)
/* void line_test (EDGE *r)
/* void check_active_edges ()
/* void rgbalong_to_bwlong (long rgbalong, long *bwlong)
/* void pixel_membership (long z, double phi)
/* void set_pixel_white (long *rgbalong)
/* void set_pixel_black (long *rgbalong)
/* void write_all_lines (long x, long y)
/* IMG_LINE *fastlines (NPSIMAGE *img)
/***********************************************************************/

/* Gradient Magnitude Threshold - for fastlines(img) */
#define THRESHOLD 30000.0

/* Gradient Angular Orientation */
#define MAX_DELTA_PHI 0.5 /* maximum difference (in radians) */

/* Constants for function void line_test (EDGE *r) */

#define MIN_PIXELS_PER_LINE 60 /* minimum pixels allowed for a IMG_LINE */
#define MIN_DMAJOR 20.0 /* minimum major axis length allowed */
#define MAX_RHO 0.1 /* maximum ratio (Rho=Dminor/Dmajor) */

#define PI 3.14159265



/* --- Global variables ---------------------------------------------*/
long Xdim; /* width of input image (nr pixels) */
long Ydim; /* width of input image (nr pixels */

int Linecount = 0; /* counter for number of IMG_LINEs made */


/* Pointers to: Present row EDGE list, Past row EDGE list, IMG_LINE list*/

EDGE *Present_edge_list_head = NULL,
       *Present_edge_list_tail = NULL,
       *Past_edge_list_head = NULL;

IMG_LINE *Line_list_head = NULL;
```

```
/*---------------------------------------------------------------*/
/* void fatal (char message)
/*
/* Prints error message and exits out of the program.
/*---------------------------------------------------------------*/
fatal(message)
   char *message;
{
 fprintf(stderr, "Fatal ERROR: ");
 perror(message);
 exit(-1); /* exit by failure */
}


/*---------------------------------------------------------------*/
/* int gradient_angles_close (double r, double s)
/*
/* Returns 1 if gradient angle orientations of EDGEs r and s
/* are within MAX_DELTA_PHI.
/* Returns 0 otherwise.
/*---------------------------------------------------------------*/
int gradient_angles_close(r,s)
   double r,s;
{
 if((r < MAX_DELTA_PHI - PI) && (s > 0.0)) r = PI + PI + r;
 else if((s < MAX_DELTA_PHI - PI) && (r > 0.0)) s = PI + PI + s;

 return(fabs(r - s) < MAX_DELTA_PHI);
}


/*---------------------------------------------------------------*/
/* int close_to_negative_pi (double phi)
/*
/* Returns 1 if orientation phi is within MAX_DELTA_PHI to -PI.
/* Returns 0 otherwise.
/*---------------------------------------------------------------*/
int close_to_negative_pi(phi)
   double phi;
{
 return(PI + phi < MAX_DELTA_PHI);
}


/*---------------------------------------------------------------*/
/* int horizontal (EDGE *e)
/*
/* Returns 1 if orientation of EDGE e (r->avg_phi) is within
/* MAX_DELTA_PHI/4 to PI/2 or -PI/2 (the normal orientations for
/* a horizontal line).
/* Returns 0 otherwise.
/*---------------------------------------------------------------*/
int horizontal(e)
  EDGE *e;
{
 double maxphi = MAX_DELTA_PHI / 4.0;

 return((fabs(e->avg_phi) > (0.5*PI)-maxphi) &&
        (fabs(e->avg_phi) < (0.5*PI)+maxphi));
}
```

```
/*----------------------------------------------------------------*/
/* EDGE *create_edge (long z, double phi)
/*
/* Returns pointer to an newly instantiated EDGE with variable
/* based upon the input pixel z and orientation phi of pixel z.
/*----------------------------------------------------------------*/
EDGE *create_edge(z, phi)
   long z; /* zth pixel in image */
   double phi; /* gradient orientation of pixel z */
{
 EDGE *r;
 long x = z%(Xdim-2), /* (x,y) coordinates of pixel z in 2D image */
 y = z/(Xdim-2);

 /* allocate memory for EDGE r */
 if((r = (EDGE *)malloc(sizeof(EDGE))) == NULL) fatal("create_edge: malloc\n"); }

 /* else initialize fields of EDGE r */
 r->active = 0;
 r->first_pixel = z;
 r->last_pixel = z;
 r->xmin = x;
 r->xmax = x;
 r->avg_phi = phi;
 r->sum_phi = phi;
 r->m00 = 1;
 r->m10 = x;
 r->m01 = y;
 r->m11 = x*y;
 r->m20 = x*x;
 r->m02 = y*y;
 r->next = NULL;

 if(Present_edge_list_head == NULL)
 Present_edge_list_head = r;
 if(Present_edge_list_tail != NULL)
 Present_edge_list_tail->next = r;
 Present_edge_list_tail = r;

 return(r);
 }
```

```
/*-------------------------------------------------------------*/
/* void add_pixel_to_edge (long z, double phi, EDGE *r)
/*
/* Updates EDGE r with new pixel z and orientation phi of pixel z.
/*-------------------------------------------------------------*/
add_pixel_to_edge(z, phi, r)
   long z; /* zth pixel */
   double phi; /* gradient orientation of pixel z */
   EDGE *r; /* EDGE to add pixel z to */
{
long   x = z%(Xdim-2), /* (x,y) coordinates of pixel z in 2D image */
       y = z/(Xdim-2);


if((r->avg_phi > 0.0) && close_to_negative_pi(phi))
{
   phi = PI + phi + PI;
}
else if((phi > 0.0) && close_to_negative_pi(r->avg_phi))
{
   r->avg_phi = PI + r->avg_phi + PI;
   r->sum_phi = r->avg_phi * r->m00;
}

/* update the new xmax for this row and last_pixel added to this EDGE */
r->xmax = x;
r->last_pixel = z;

/* update the least squares fit moments */
++r->m00;
r->m10 += x;
r->m01 += y;
r->m11 += x*y;
r->m20 += x*x;
r->m02 += y*y;

/* recalculate the average gradient orientation for the EDGE */
r->sum_phi += phi;
r->avg_phi = r->sum_phi / r->m00;
}
```

```
/*------------------------------------------------------------*/
/* EDGE *combine_edges (EDGE *r1, EDGE *r2)
/*
/* Returns pointer to EDGE r1 after appending all information
/* of EDGE r2 with r1.
/*------------------------------------------------------------*/
EDGE *combine_edges(r1,r2)
   EDGE *r1, *r2;
{

 /* Special case where r1->first_pixel is changed: */
 if(!((r2->first_pixel > r1->first_pixel) || (horizontal(r1) && (r1->xmin < r2->xmin))))
 {
   r1->first_pixel = r2->first_pixel;
 }


 /* Modify last_pixel */
 if(r2->last_pixel > r1->last_pixel)
 {
   r1->last_pixel = r2->last_pixel;
   r1->xmax = r2->xmax;
 }
 else if(horizontal(r2) && (r2->xmax > r1->xmax))
 {
   r1->last_pixel = r2->last_pixel;
 }


 /* Special case: if gradient angles on opposite sides of +/- pi */
 if((r1->avg_phi > 0.0) && close_to_negative_pi(r2->avg_phi))
 {
   r2->avg_phi = PI + r2->avg_phi + PI;
   r2->sum_phi = r2->avg_phi * r2->m00;
 }
 else if((r2->avg_phi > 0.0) && close_to_negative_pi(r1->avg_phi))
 {
   r1->avg_phi = PI + r1->avg_phi + PI;
   r1->sum_phi = r1->avg_phi * r1->m00;
 }

 /* combine least squares fit moments */
 r1->m00 += r2->m00;
 r1->m10 += r2->m10;
 r1->m01 += r2->m01;
 r1->m11 += r2->m11;
 r1->m20 += r2->m20;
 r1->m02 += r2->m02;

 /* combine and recalculate average gradient orientation for the EDGE */
 r1->sum_phi += r2->sum_phi;
 r1->avg_phi = r1->sum_phi / r1->m00;

 return(r1); /* return EDGE r1 with combined information of r1 & r2 */
}
```

```
/*--------------------------------------------------------------*/
/* IMG_LINE *create_line (EDGE *r, double M20, double M11, double M02,
/* double Dmajor, double Dminor, double Rho)
/*
/* Returns pointer to newly instantiated IMG_LINE with variables set
/* according to moments described by EDGE r, secondary moments
/* M20, M11, and M02, axis lengths Dmajor, Dminor, and ratio of
/* axis lengths Rho.
/*--------------------------------------------------------------*/

IMG_LINE *create_line(r,M20,M11,M02,Dmajor,Dminor,Rho)
   EDGE *r;
   double M20,M11,M02,Dmajor,Dminor,Rho;

{

 IMG_LINE *l;

 /* r->first_pixel mapped onto the IMG_LINE will be endpoint p1
 r->last_pixel mapped onto the IMG_LINE will be endpoint p2 */

 long x1 = r->first_pixel%(Xdim-2),
    y1 = r->first_pixel/(Xdim-2),
    x2 = r->last_pixel%(Xdim-2),
    y2 = r->last_pixel/(Xdim-2);


 /* Calculate the normal orientation of the IMG_LINE by atan2() function. */

 double Phi = atan2(-2*M11,M02-M20)/2.0,

 /* Delta1 and delta2 are the offsets used to calculate the endpoints
 for the IMG_LINE segment based upon values x1,y1 and x2,y2. */

 delta1 = (r->m10/r->m00 - (double)x1)*fcos(Phi) +
    (r->m01/r->m00 - (double)y1)*fsin(Phi),
 delta2 = (r->m10/r->m00 - (double)x2)*fcos(Phi) +
    (r->m01/r->m00 - (double)y2)*fsin(Phi);


 /* Phi = atan2(-2*M11,M02-M20)/2.0) always returns positive result to Phi.
 Therefore, negative_phi = (r->avg_phi < 0.0) is necessary. */

 int negative_phi = (r->avg_phi < 0.0);

 /* Allocate memory for IMG_LINE l. */
 if((l = (IMG_LINE *)malloc(sizeof(IMG_LINE))) == NULL) fatal("create_line: malloc\n");


 /* Calculate x,y coordinates for endpoints p1 and p2. */

 l->p1.x = (double)x1 + delta1*fcos(Phi);
 l->p1.y = (double)y1 + delta1*fsin(Phi);
 l->p2.x = (double)x2 + delta2*fcos(Phi);
 l->p2.y = (double)y2 + delta2*fsin(Phi);
```

```
/* Copy least squares fit moments. */

l->m00 = r->m00;
l->m10 = r->m10;
l->m01 = r->m01;
l->m11 = r->m11;
l->m20 = r->m20;
l->m02 = r->m02;


/* Phi is positive, but -pi < r->avg_phi < pi. */

if(negative_phi) l->phi = -Phi;
else l->phi = Phi;


/* Update rest of IMG_LINE values. */

l->next = NULL;

l->dmajor = Dmajor;
l->dminor = Dminor;
l->rho = Rho;

l->matchlist = NULL;
l->pm = NULL;


++Linecount; /* Increment global variable, Linecount. */
strcpy(l->name," ");
sprintf(l->name, "%d", Linecount);
return(l); /* Return IMG_LINE l. */

}
```

```
/*-----------------------------------------------------------*/
/* void line_test (EDGE *r)
/*
/* Determines if EDGE r meets three requirements to be a IMG_LINE:
/* (1) The number of pixels in EDGE r (r->m00) be greater than MIN_PIXELS_PER_LINE.
/* (2) The ratio (Rho) of the length of major and minor axes of the
/* EDGE be less than MAX_RHO.
/* (3) The length of the major axis (Dmajor) be greater than MIN_DMAJOR, the minimum
/* IMG_LINE length allowed.
/* If all three conditions are met, a new IMG_LINE type is created and
/* appended to the Line_list in order of significance (in this case Dmajor).
/*-----------------------------------------------------------*/
line_test(r)
   EDGE *r;
{
 IMG_LINE *l, *insert_pt = Line_list_head;
 double M20,M11,M02,Ma,Mb,Mmajor,Mminor,Dmajor,Dminor,Rho;

 /* First test -- A IMG_LINE must have a required minimun number of pixels. */
 if(r->m00 > MIN_PIXELS_PER_LINE)
 {
   /* Calculate secondary moments by least squares fit. */
   M20 = r->m20 - ((r->m10*r->m10)/r->m00);
   M11 = r->m11 - ((r->m10*r->m01)/r->m00);
   M02 = r->m02 - ((r->m01*r->m01)/r->m00);

   /* Calculate major and minor axis lengths, Dmajor and Dminor. */
   Ma = (M20+M02)/2.0;
   Mb = sqrt( ((M02-M20)*(M02-M20)/4.0) + (M11*M11) );
   Mmajor = Ma - Mb;
   Mminor = Ma + Mb;
   Dmajor = 4.0*sqrt(Mminor/r->m00);
   Dminor = 4.0*sqrt(Mmajor/r->m00);

   /* Calculate ratio Rho. */
   Rho = Dminor/Dmajor;

  /* Second & Third tests -- Ratio Rho must represent a line, not a blob.
   -- IMG_LINE must be at least a certain length. */
   if((Rho < MAX_RHO) && (Dmajor > MIN_DMAJOR))
   {
       /* The EDGE passed the three requirments to be a line. */
       l = create_line(r,M20,M11,M02,Dmajor,Dminor,Rho);

       /* Add new IMG_LINE to IMG_LINE list in order by IMG_LINE length, dmajor. */
       if(Line_list_head == NULL) Line_list_head = l;
       else if(l->dmajor > Line_list_head->dmajor)
       {
           l->next = Line_list_head;
           Line_list_head = l;
       }
       else
       {
           while((insert_pt->next != NULL) && (l->dmajor < insert_pt->next->dmajor))
               insert_pt = insert_pt->next;
           l->next = insert_pt->next;
           insert_pt->next = l;
       }
   } /* end if second and third tests */
 } /* end if first test */
}
```

```
/*--------------------------------------------------------------*/
/* void check_active_edges ()
/*
/* This function, performed at the start of scanning for each row,
/* cycles through the EDGEs in the Present_edge_list to see if:
/* (1) Adajacent EDGEs on the same row have gradient orientations
/* that are close and therefore can be combined into one
/* EDGE.
/* (2) EDGEs from the Present_edge_list are adjacent to
/* EDGEs from the past row's Previous_edge_list and their
/* gradient orientations are close so that they may be
/* combined into one EDGE.
/* The second method for determining when to combine_edges()
/* provides the means for constructing EDGEs across rows.
/*
/* All EDGEs not combined with a EDGE from the past row are
/* tested by line_test() which examines the EDGE for satisfying
/* the IMG_LINE requirements and appends the IMG_LINE to the Line_list.
/*--------------------------------------------------------------*/
check_active_edges()
{
 EDGE *pres = Present_edge_list_head, /* the EDGEs found by the scan of this row */
        *past = Past_edge_list_head, /* the EDGEs found during the scan of the previous row */
        *temp; /* a temporary pointer used for freeing memory */

 int continue_loop; /* a boolean integer */


 /* Loop through all EDGEs in the Present_edge_list (all EDGEs found
   during the scan of the present row. */

 while(pres != NULL)
 {

 /* Look forward to the next EDGEs on the present row to see if
   any should be combined with this EDGE. If so, combine_edges()
   and free() the second EDGE. */

   while((pres->next != NULL) && (pres->xmax+1 == pres->next->xmin) &&
           gradient_angles_close(pres->avg_phi,pres->next->avg_phi))
      {
           pres = combine_edges(pres,pres->next);
           temp = pres->next;
           pres->next = pres->next->next;
           free(temp);
      }

 /* Set continue_loop boolean to true and loop through the EDGEs of
   found during the scan of the previous row of pixels. */
   continue_loop = 1;

   while((past != NULL) && (pres->xmax > past->xmin-2) && (continue_loop))
      {

           /* If the two EDGEs are adjacent and their gradient orientations
           are close, then the information of the past EDGE must be in-
           cluded with the present EDGE and the EDGE should be marked
           that it is still "active". */
```

```
        if((pres->xmin < past->xmax+2) && gradient_angles_close(past->avg_phi,pres->avg_phi))
        {
            pres = combine_edges(pres,past);
            past->active = 1;
        }

            /* If the past EDGE was not appended to a present EDGE, (i.e.
            not "active") then no more pixels may be added to it and it
            shall be tested if it satisfies the conditions for an IMG_LINE. */

            if((past->next != NULL) && (pres->xmax > past->next->xmin-2))
            {
                if(!past->active) line_test(past);
                past = past->next;
            }

            /* If the past EDGE did not meet any of the above two conditions,
            then exit this loop and continue evaluation with the next
            EDGE of the present row. */

            else continue_loop = 0;
        }

    pres = pres->next;
    }


/* Continue line_test() and free() all EDGEs in the past row. */
while(past != NULL)
{
    temp = past;
    if(!past->active) line_test(past);
    past = past->next;
    free(temp);
}


/* Finally: Past row <- Present row, Present row <- empty. */
Past_edge_list_head = Present_edge_list_head;
Present_edge_list_head = NULL;
Present_edge_list_tail = NULL;
}
```

```
/*--------------------------------------------------------------*/
/* void rgbalong_to_bwlong (long rgbalong, long *bwlong)
/*
/* Converts color to black/white for rgba formatted pixels.
/* The weights assigned for each color are television standards.
/* This function courtesy of M. Zyda.
/*--------------------------------------------------------------*/

rgbalong_to_bwlong(rgbalong,bwlong)
   long rgbalong; /* input color rgbalong */
   long *bwlong; /* output b/w rgbalong */
{
 unsigned char red, green, blue, alpha;
 unsigned bw;

 /* Use bit masks to get RGB and alpha values from input rgbalong. */
 red = rgbalong & 0x000000ff;
 green = (rgbalong & 0x0000ff00) >> 8;
 blue = (rgbalong & 0x00ff0000) >> 16;
 alpha = (rgbalong & 0xff000000) >> 24;

 /* Calculate the black&white intesity using NTSC standard.
 intensity = 0.299(red) + 0.587(green) + 0.114(blue) */
 bw = (0.299*red)+(0.587*green)+(0.114*blue);

 /* Save the black&white intensity in bwlong. */
 *bwlong = (alpha<<24)|(bw<<16)|(bw<<8)|bw;
 }



/*--------------------------------------------------------------*/
/* void pixel_membership (long z, double phi)
/*
/* For a given row while scanning the black&white image. (scan of row> ....yyyyy....rrrrrrrrz
/* y = pixels included in previous EDGE of same row,
/* r = pixels included in EDGE r,
/* z = pixel z being tested for inclusion with EDGE r)
/*
/* two requirements must be satisfied for a pixel (z) with the pixels of a given EDGE:
/* (1) pixels must be adjacent (i.e. the last_pixel of the EDGE + 1 must equal pixel z),
/* (2) the gradient angle orientations of the last_pixel and z must be "close". Otherwise, pixel z
/* can not be included with EDGE r and must be considered the first_pixel of a new EDGE for this row.
/*--------------------------------------------------------------*/
pixel_membership(z, phi)
   long z; /* the zth pixel of the gradient image */
    double phi;
{
 EDGE *r;

 if((Present_edge_list_tail != NULL) &&
    (Present_edge_list_tail->last_pixel+1 == z) &&
    gradient_angles_close(Present_edge_list_tail->avg_phi,phi))
    {
         add_pixel_to_edge(z,phi,Present_edge_list_tail);
    }
 else /* create new EDGE and add to end of Present list */
    {
         r = create_edge(z,phi);
    }
}
```

```
/*-----------------------------------------------------------------*/
/* void set_pixel_white (long *rgbalong)
/*
/* Sets the specified long integer pointed to by rgbalong to be
/* white by setting all RGB bits to ff (255 dec -> max intensity).
/*-----------------------------------------------------------------*/
set_pixel_white(rgbalong)
    long *rgbalong;
{
 *rgbalong = 0xffffffff;
}



/*-----------------------------------------------------------------*/
/* void set_pixel_black (long *rgbalong)
/*
/* Sets the specified long integer pointed to by rgbalong to be
/* black by setting all RGB bits to 00 (0 dec -> min intensity).
/*-----------------------------------------------------------------*/
set_pixel_black(rgbalong)
    long *rgbalong;
{
 *rgbalong = 0xff000000;
}



/*-----------------------------------------------------------------*/
/* void write_all_lines (long x, long y)
/*
/* Write to output file "name.text".
/*-----------------------------------------------------------------*/
write_all_lines(x,y)
    long x,y; /* the x,y dimensions of the image */
{
 IMG_LINE *l = Line_list_head;
 FILE *lines_file;


 lines_file = fopen("lines.text","w");

 fprintf(lines_file,"%d %d\n\n",x,y);

 while(l!=NULL)
 {
   fprintf(lines_file,"%.2f %.2f\n%.2f %.2f\n%.4f\n",
   l->p1.x,l->p1.y,l->p2.x,l->p2.y,l->phi);
   l = l->next;
 }

 fclose(lines_file);

 printf(" lines found in image written to: 'lines.text'\n");
}
```

```
/*------------------------------------------------------------*/
/* IMG_LINE *fastlines(NPSIMAGE)
/*
/* Returns a pointer to the head of a linked list of vertical IMG_LINEs found in an NPSIMAGE.
/*------------------------------------------------------------*/
IMG_LINE *fastlines(img)
   NPSIMAGE *img; /* input file_name.rgb color image */
{
 long *ptr; /* pointer to bitsptr of NPSIMAGE */
 long grmask = 0x0000ff00;
 double dx, dy, Th = THRESHOLD*THRESHOLD;
 register int i = 0, /* counter for pixels in input image */
              z = 0; /* counter for pixels in gradient image */
EDGE *reg;

   Xdim = img->xsize;
   Ydim = img->ysize;
   ptr = img->imgdata.bitsptr;
   Linecount = 0;

 /* The scan of an RGB image is from the bottom row -> up, traversing the rows left to right. */
for(i = Xdim + 1; i < (Xdim*(img->ysize-1))-1; ++i)
 {

   /* If pixel i is in the leftmost column, do check_active_edges(). */
   if(i%Xdim == 0) check_active_edges();

   /* Ensure pixel i is not in leftmost or rightmost column */
   else if(i%Xdim != Xdim-1)
        {
        /* Calculate dx,dy via Sobel operator for pixel i. */

        dx = (-(ptr[i+Xdim-1]&grmask) + (ptr[i+Xdim+1]&grmask)
             -(2 * (ptr[i-1]&grmask))+ (2 * (ptr[i+1]&grmask))
             -(ptr[i-Xdim-1]&grmask) + (ptr[i-Xdim+1]&grmask));

        dy = ( (ptr[i+Xdim-1]&grmask) + (2*(ptr[i+Xdim]&grmask))
             + (ptr[i+Xdim+1]&grmask) - (ptr[i-Xdim-1]&grmask)
             - (2*(ptr[i-Xdim]&grmask)) - (ptr[i-Xdim+1]&grmask));

        if((dx*dx)+(dy*dy) > Th) pixel_membership(z,atan2(dy,dx));
        ++z;
        }

 } /* endfor i */

 /* Check remaining EDGEs for lines: */
 reg = Past_edge_list_head;
 while(reg != NULL)
 {
   line_test(reg);
   reg = reg->next;
 }

 /* Write the lines list to file "lines.text". */
 write_all_lines(img->xsize,img->ysize);

 printf(" Number lines found in %s = %d\n",img->name,Linecount);

 return(Line_list_head);
}
```

F. FILE: DISPLAYSUPPORT.H

```
/**********************************************************************/
/* FILENAME: displaysupport.h
/* AUTHOR: Kevin Peterson
/* DATE: 31 January 1992
/*DESCRIPTION: Collection of display functions.
/*
/* void display_bw_and_gradient_images (NPSIMAGE *img1, NPSIMAGE *img2,
/*                                        IMG_LINE *l)
/* void draw_red_lines (IMG_LINE *l)
/* void display_loop (NPSIMAGE *img1, long winid1,
/*                     NPSIMAGE *img2, long winid2)
//* void display_line_image (NPSIMAGE *img, IMG_LINE *l)
/* void display_line_loop (NPSIMAGE *img, IMG_LINE *l, long winid)
/*
/* These functions make calls to following SiliconGraphics routines:
/* prefsize(), winopen(), winset(), winclose(), RGBmode(),
/* singlebuffer(), gconfig(), qdevice(), c3f(), move2(), draw2(),
/* swapbuffers(), lrectwrite(), reshapeviewport(), and clear().
/**********************************************************************/
/*------------------------------------------------------------*/
/* void display_bw_and_gradient_images (NPSIMAGE *img1, NPSIMAGE *img2,
/*                                        IMG_LINE *l)
/* Displays NPSIMAGE img1 and img2 on SiliconGraphics' Iris workstation.
/* Each window will be displayed within red outline when operator depresses left mouse button.
/*------------------------------------------------------------*/
display_bw_and_gradient_images(img1,img2,l)
   NPSIMAGE     *img1, /* input b/w image */
                *img2; /* gradient image */
   IMG_LINE *l;
{
 long winid1, winid2; /* silicon graphics window id's */

 prefsize(img1->xsize,img1->ysize); /* preferred size for window */
 winid1 = winopen(img1->name); /* open the window */
 RGBmode(); /* set RGBmode, singlebuffer, and */
 singlebuffer(); /* configure the window */
 gconfig();
prefsize(img2->xsize,img2->ysize); /* preferred size for window */
 winid2 = winopen(img2->name); /* open the window */
 RGBmode(); /* set RGBmode, singlebuffer, and */
 singlebuffer(); /* configure the window */
 gconfig();

 /* initialize_controls */
 qdevice(REDRAW);
 qdevice(LEFTMOUSE);
 qdevice(MIDDLEMOUSE);
 qdevice(RIGHTMOUSE);
 qdevice(ESCKEY);

 display_loop(img1,winid1,img2,winid2,l);

 free(img1->imgdata.bitsptr); /* delete the bitmap for the image */
 free(img1); /* delete the NPSIMAGE structure */
 winclose(winid1); /* close the window */
 free(img2->imgdata.bitsptr);
 free(img2);
 winclose(winid2);
}
```

```
/*-------------------------------------------------------------*/
/* void draw_red_lines(IMG_LINE *l)
/*
/* Draws red lines over an image.
/*-------------------------------------------------------------*/
draw_red_lines(l)
   IMG_LINE *l;
{
 static float red[3] = {1.0,0.0,0.0}; /* rgb red */

 c3f(red);
 while(l!=NULL)
 {
   move2(l->p1.x,l->p1.y);
   draw2(l->p2.x,l->p2.y);
   l = l->next;
 }
 swapbuffers();
}


/*-------------------------------------------------------------*/
/* void draw_white_lines(IMG_LINE *l)
/*
/* Draws white lines over an image.
/*-------------------------------------------------------------*/
draw_white_lines(l)
   IMG_LINE *l;
{
 static float white[3] = {1.0,1.0,1.0}; /* rgb white */

 c3f(white);
 while(l!=NULL)
 {
   move2(l->p1.x,l->p1.y);
   draw2(l->p2.x,l->p2.y);
   l = l->next;
 }
 swapbuffers();
}


/*-------------------------------------------------------------*/
/* void draw_black_lines(IMG_LINE *l)
/*
/* Draws black lines over an image.
/*-------------------------------------------------------------*/
draw_black_lines(l)
   IMG_LINE *l;
{
 static float black[3] = {0.0,0.0,0.0}; /* rgb black */

 c3f(black);
 while(l!=NULL)
 {
   move2(l->p1.x,l->p1.y);
   draw2(l->p2.x,l->p2.y);
   l = l->next;
 }
 swapbuffers();
}
```

```
/*-----------------------------------------------------------*/
/* void display_loop (NPSIMAGE *img1, long winid1,
/* NPSIMAGE *img2, long winid2)
/*
/* Continuously displays the white&white (img1) and gradient (img2)
/* images within their respective windows (winid1 and winid2).
/* - Depress left mouse button when cursor arrow is in title
/*  bar to see lines initially over the image.
/* - Depress middle mouse button when cursor is in image to
/*  see lines only.
/* - Depress middle mouse button when cursor is in title bar
/*  "drag" window and redisplay image with lines.
/* - Depress right or left mouse buttons when cursor is i
/* either image to kill windows.
/*-----------------------------------------------------------*/
display_loop(img1,winid1,img2,winid2,l)
   NPSIMAGE *img1, /* input b/w image */
                  *img2; /* gradient image */
    long winid1, winid2; /* window id's for images */
    IMG_LINE *l;

{
static float white[3] = {1.0,1.0,1.0}; /* rgb white */
short value; /* value returned from the event queue */

/* display the images once */
winset(winid1);
lrectwrite(0,0,img1->xsize - 1,img1->ysize - 1,img1->imgdata.bitsptr);
winset(winid2);
lrectwrite(0,0,img2->xsize - 1,img2->ysize - 1,img2->imgdata.bitsptr);

/* loop until a mouse button is pressed */
while(TRUE)
{
switch(qread(&value))
  {
  case REDRAW:
        winset((long)value);
        reshapeviewport();
        if(value == winid1) lrectwrite(0,0,img1->xsize - 1,img1->ysize - 1, img1->imgdata.bitsptr);
        if(value == winid2) lrectwrite(0,0,img2->xsize - 1,img2->ysize - 1, img2->imgdata.bitsptr);
        draw_red_lines(l);
        break;
  case LEFTMOUSE:
  case MIDDLEMOUSE:
        if(value == 0) {
             c3f(white);
             clear();
             draw_red_lines(l);
        }
        break;
  case RIGHTMOUSE:
        if(value == 0) return;
        break;
  case ESCKEY:
        exit(0);
  default:
        break;
  } /* end switch */
 } /* end while */
}
```

```
/*----------------------------------------------------------------*/
/* void display_line_image (NPSIMAGE *img, IMG_LINE *l)
/*----------------------------------------------------------------*/
display_line_image(img,l)
   NPSIMAGE *img;
   IMG_LINE *l;
{
 long winid;

 prefsize(img->xsize,img->ysize); /* preferred size for window */
 winid = winopen(img->name); /* open the window */
 RGBmode(); /* set RGBmode, singlebuffer, and */
 singlebuffer(); /* configure the window */
 gconfig();

 /* initialize_controls */
 qdevice(REDRAW);
 qdevice(LEFTMOUSE);
 qdevice(MIDDLEMOUSE);
 qdevice(RIGHTMOUSE);
 qdevice(ESCKEY);

 display_line_loop(img,l,winid);

 free(img->imgdata.bitsptr); /* delete the bitmap for the image */
 free(img); /* delete the NPSIMAGE structure */
 winclose(winid); /* close the window */
 }




/*----------------------------------------------------------------*/
/* void display_line_loop (NPSIMAGE *img, IMG_LINE *l, long winid)
/*----------------------------------------------------------------*/
display_line_loop(img,l,winid)
   NPSIMAGE *img;
   IMG_LINE *l;
   long winid;

{
 static float white[3] = {1.0,1.0,1.0}; /* rgb white */
 short value; /* value returned from the event queue */

 /* display the images once */
 winset(winid);
 lrectwrite(0,0,img->xsize - 1,img->ysize - 1,img->imgdata.bitsptr);

 /* loop until a mouse button is pressed */
 while(TRUE)
 {

 switch(qread(&value))
   {

   case REDRAW:
         winset((long)value);
         reshapeviewport();
         if(value == winid) lrectwrite(0,0,img->xsize - 1,img->ysize - 1, img->imgdata.bitsptr);
         break;
```

```
    case LEFTMOUSE:
        /* draw white lines over input image */
        if(value == 0) draw_white_lines(l);
        break;

    case MIDDLEMOUSE:
        if(value == 0)
        {
            c3f(white);
            clear();
            draw_black_lines(l);
        }
        break;

    case RIGHTMOUSE:
        if(value == 0) return;
        break;

    case ESCKEY:
        exit(0);

    default:
        break;

    } /* end switch */

  } /* end while */

}
```

G. FILE: VERSUPPORT.H

```
/**********************************************************************/
/* FILENAME: vertsupport.h
/* AUTHOR: Kevin Peterson
/* DATE: 29 January 1992
/*
/*DESCRIPTION: Supplementary vertical edge finding functions.
/* int vertical(EDGE)
/* void vertical_line_test(EDGE)
/* void check_active_vertical_edges()
/* IMG_LINE *vertlines(NPSIMAGE)
/**********************************************************************/

#define VERT_CONSTRAINT 0.0349 /* radians (= 2.0 degrees) */


/*------------------------------------------------------------------*/
/* int vertical(EDGE)
/*
/* Returns 1 if orientation of EDGE r (r->avg_phi) is within
/* MAX_DELTA_PHI/4 of 0, PI, or -PI (the normal orientations for
/* a vertical line).
/* Returns 0 otherwise.
/*------------------------------------------------------------------*/
int vertical(r)
 EDGE *r;
{
  return ((fabs(r->avg_phi) < VERT_CONSTRAINT) ||
    (fabs(r->avg_phi) > PI - VERT_CONSTRAINT));
}
```

```
/*---------------------------------------------------------------*/
/* void vertical_line_test(EDGE)
/*
/* Same as line_test(r), but also ensures that EDGE r is vertical.
/*---------------------------------------------------------------*/
vertical_line_test(r)
   EDGE *r;
{
IMG_LINE *l, *insert_pt = Line_list_head;
double M20,M11,M02,Ma,Mb,Mmajor,Mminor,Dmajor,Dminor,Rho;

 /* First test -- A IMG_LINE must have a required minimun number of pixels. */
 if((r->m00 > MIN_PIXELS_PER_LINE) && (vertical(r)))
 {
   /* Calculate secondary moments by least squares fit. */
   M20 = r->m20 - ((r->m10*r->m10)/r->m00);
   M11 = r->m11 - ((r->m10*r->m01)/r->m00);
   M02 = r->m02 - ((r->m01*r->m01)/r->m00);

   /* Calculate major and minor axis lengths, Dmajor and Dminor. */
   Ma = (M20+M02)/2.0;
   Mb = sqrt( ((M02-M20)*(M02-M20)/4.0) + (M11*M11) );
   Mmajor = Ma - Mb;
   Mminor = Ma + Mb;
   Dmajor = 4.0*sqrt(Mminor/r->m00);
   Dminor = 4.0*sqrt(Mmajor/r->m00);

   /* Calculate ratio Rho. */
   Rho = Dminor/Dmajor;

  /* Second & Third tests -- Ratio Rho must represent a line, not a blob.
   -- IMG_LINE must be at least a certain length. */
   if((Rho < MAX_RHO) && (Dmajor > MIN_DMAJOR))
       {
        /* The EDGE passed the four requirments to be a vertical line. */
        l = create_line(r,M20,M11,M02,Dmajor,Dminor,Rho);

        /* Add new IMG_LINE to IMG_LINE list in order by
        IMG_LINE length, dmajor. */
        if(Line_list_head == NULL) Line_list_head = l;
        else if(l->dmajor > Line_list_head->dmajor)
        {
            l->next = Line_list_head;
            Line_list_head = l;
        }
        else
        {
        while((insert_pt->next != NULL) &&
            (l->dmajor < insert_pt->next->dmajor))
            {
                insert_pt = insert_pt->next;
            }
        l->next = insert_pt->next;
        insert_pt->next = l;
        }

   } /* end if second and third tests */

 } /* end if first test */
}
```

```
/*-----------------------------------------------------------------*/
/* void check_active_vertical_edges()
/*
/* Same as check_active_edges(), but for vertical EDGEs only.
/*-----------------------------------------------------------------*/
check_active_vertical_edges()
{
EDGE *pres = Present_edge_list_head,
/* the EDGEs found by the scan of this row */
*past = Past_edge_list_head,
/* the EDGEs found during the scan of the previous row */
*temp; /* a temporary pointer used for freeing memory */

int continue_loop; /* a boolean integer */

/* Loop through all EDGEs in the Present_edge_list (all EDGEs
found during the scan of the present row. */

while(pres != NULL)
{
   /* Look forward to the next EDGEs on the present row to see if
      any should be combined with this EDGE. If so, combine_edges()
      and free() the second EDGE. */

   while((pres->next != NULL) && (pres->xmax+1 == pres->next->xmin) &&
      gradient_angles_close(pres->avg_phi,pres->next->avg_phi))
      {
         pres = combine_edges(pres,pres->next);
         temp = pres->next;
         pres->next = pres->next->next;
         free(temp);
      }

   /* Set continue_loop boolean to true and loop through the EDGEs of
      found during the scan of the previous row of pixels. */

   continue_loop = 1;
   while((past != NULL) && (pres->xmax > past->xmin-2) && (continue_loop))
      {

      /* If the two EDGEs are adjacent and their gradient orientations
      are close, then the information of the past EDGE must be in-
      cluded with the present EDGE and the EDGE should be marked
      that it is still "active". */

      if((pres->xmin < past->xmax+2) &&
      gradient_angles_close(past->avg_phi,pres->avg_phi))
      {
         pres = combine_edges(pres,past);
         past->active = 1;
      }

      /* If the past EDGE was not appended to a present EDGE, (i.e.
      not "active") then no more pixels may be added to it and it
      shall be tested if it satisfies the conditions for an IMG_LINE. */

      if((past->next != NULL) && (pres->xmax > past->next->xmin-2))
      {
         if(!past->active) vertical_line_test(past);
         past = past->next;
      }
```

```
        /* If the past EDGE did not meet any of the above two conditions,
        then exit this loop and continue evaluation with the next
        EDGE of the present row. */

        else continue_loop = 0;
        }

        pres = pres->next;
}

/* Continue vertical_line_test() and free() all EDGEs in the past row. */
while(past != NULL)
{
   temp = past;
   if(!past->active) vertical_line_test(past);
   past = past->next;
   free(temp);
}

/* Finally: Past row <- Present row, Present row <- empty. */
Past_edge_list_head = Present_edge_list_head;
Present_edge_list_head = NULL;
Present_edge_list_tail = NULL;
}
```

```
/*----------------------------------------------------------------*/
/* IMG_LINE *vertlines(NPSIMAGE)
/*
/* Returns a pointer to the head of a linked list of vertical
/* IMG_LINEs found in an NPSIMAGE.
/*----------------------------------------------------------------*/
IMG_LINE *vertlines(img)
   NPSIMAGE *img; /* input file_name.rgb color image */
{
 long *ptr; /* pointer to bitsptr of NPSIMAGE */
 long grmask = 0x0000ff00;
 double dx, dy, Th = THRESHOLD*THRESHOLD;
 register int i = 0, /* counter for pixels in input image */
 z = 0; /* counter for pixels in gradient image */
 EDGE *reg;

Xdim = img->xsize;
Ydim = img->ysize;
ptr = img->imgdata.bitsptr;
Linecount = 0;


 /* The scan of an RGB image is from the bottom row -> up,
 traversing the rows left to right. */

 for(i = Xdim + 1; i < (Xdim*(img->ysize-1))-1; ++i)
 {
   /* If pixel i is in the leftmost column, do check_active_edges(). */
   if(i% Xdim == 0) check_active_vertical_edges();

   /* Ensure pixel i is not in leftmost or rightmost column */
   else if(i% Xdim != Xdim-1)
       {
       /* Calculate dx,dy via Sobel operator for pixel i. */

       dx = (-(ptr[i+Xdim-1]&grmask) + (ptr[i+Xdim+1]&grmask)
           -(2 * (ptr[i-1]&grmask))+ (2 * (ptr[i+1]&grmask))
           -(ptr[i-Xdim-1]&grmask) + (ptr[i-Xdim+1]&grmask));

       dy = ( (ptr[i+Xdim-1]&grmask) + (2*(ptr[i+Xdim]&grmask))
           + (ptr[i+Xdim+1]&grmask) - (ptr[i-Xdim-1]&grmask)
           - (2*(ptr[i-Xdim]&grmask)) - (ptr[i-Xdim+1]&grmask));

       if((dx*dx)+(dy*dy) > Th) pixel_membership(z,atan2(dy,dx));
       ++z;
       }
 } /* endfor i */

 /* Check remaining EDGEs for lines: */
 reg = Past_edge_list_head;
 while(reg != NULL)
 {
   vertical_line_test(reg);
   reg = reg->next;
 }

 /* Write the lines list to file "lines.text". */
 write_all_lines(img->xsize,img->ysize);
printf("Number lines found in %s = %d\n",img->name,Linecount);
return(Line_list_head);
}
```

APPENDIX C - LINE MATCHING AND POSE DETERMINATION ROUTINES

A. IMPLEMENTATION: VERTMATCH.C

```
/***********************************************************************/
/* FILENAME: vertmatch.c
/* AUTHOR: Kevin Peterson
/* DATE: 09 March 1992
/***********************************************************************/
#include <gl.h> /* SiliconGraphics (r) graphic library */
#include <gl/image.h> /* SGI image structure library */
#include <device.h> /* Machine-dependent device library for keys and mouse-buttons */
#include <stdio.h> /* C standard i/o library */
#include <math.h> /* C math library for atan2() */

#include "2d+.h" /*Header files for 3D model environment by J. Stein */
#include "5th.h"
#include "graphics.h"
#include "visibility.h"

#include "image_types.h" /* Type definitions for NPSIMAGE, etc. */
#include "match_types.h" /* Type definitions for EDGE, IMG_LINE.. */

#include "npsimagesupport.h" /* Some NPSIMAGE functions */
#include "edgesupport.h" /* EDGE and IMG_LINE building functions */
#include "vertsupport.h" /* Vertical EDGE and IMG_LINE supplement */
#include "matchsupport.h" /* LINE and IMG_LINE matching routines */
#include "matchdisplaysupport.h" /* Graphics display functions */


main(argc, argv)
        int argc;
        char *argv[];
{
 WORLD *FifthFloor;
 NPSIMAGE *img;
 IMG_LINE *ImageLinesHead = NULL, *imageline;
 LINE_HEAD *ModelLinesHead1 = NULL, *ModelLinesHead2 = NULL;
 POSE *in_pose, *corr_pose;
 float x0, y0, theta0, u_posit, center_u_axis;

if (argc != 2) fatal("usage: vertmatch filename\n");

 /* Pose values for data images: ---------------------------- */

 x0 = 60.0; y0 = 366.0; theta0 = 250.0; /* 060366250.pic */
 /* x0 = 60.0; y0 = 366.0; theta0 = 253.0; /* 060366253.pic */
 /* x0 = 60.0; y0 = 366.0; theta0 = 251.0; /* 060366251.pic */
 /* x0 = 59.0; y0 = 366.0; theta0 = 250.0; /* 059366249.pic */
 /* x0 = 94.0; y0 = 381.0; theta0 = 245.0; /* 094381245.pic */
 /* x0 = 68.0; y0 = 372.0; theta0 = 249.0; /* 068372249.pic */

 /* x0 = 48.0; y0 = 277.0; theta0 = 000.0; /* 048277000.pic */
 /* x0 = 48.0; y0 = 277.0; theta0 = 002.0; /* 048277002.pic */
 /* x0 = 48.0; y0 = 277.0; theta0 = 356.0; /* 048277356.pic */
 /* x0 = 43.0; y0 = 277.0; theta0 = 356.0; /* 043277356.pic */
 /* x0 = 43.0; y0 = 267.0; theta0 = 000.0; /* 043267000.pic */
 /* x0 = 48.0; y0 = 267.0; theta0 = 000.0; /* 048267000.pic */
 /* x0 = 48.0; y0 = 272.0; theta0 = 000.0; /* 048272000.pic */
```

```
/* Read input rgb image and extract vertical edges. */
 img = read_sgi_rgbimage(argv[1]);
 if(img ==(NPSIMAGE *) NULL) fatal("File %s is a NULL image.\n",img->name);
 printf("vertmatch> %s xsize= %d ysize= %d pixels= %d\n",
                img->name,img->xsize,img->ysize, (img->xsize*img->ysize));

 ImageLinesHead = vertlines(img);
 imageline = ImageLinesHead;
 center_u_axis = (float)(img->xsize/2);
 while(imageline != NULL)
 {
           u_posit = (imageline->p1.x + imageline->p2.x)/2.0;
           imageline->angle_to_image_center = atan2((center_u_axis - u_posit),
                                    FOCAL_LENGTH_IN_PIXELS);
           imageline = imageline->next;
 }

/* Declare the estimated pose of the camera. */
 if((in_pose = (POSE *)malloc(sizeof(POSE))) == NULL)
           fatal("creating POSE in_pose: malloc\n");
 in_pose->x = x0;
 in_pose->y = y0;
 in_pose->theta = normalize(theta0 * (PI/180.0));
 printf("\nDR (input) pose: x = %.2f, y = %.2f, theta = %.2f(%.2f rads)\n\n",
           in_pose->x,in_pose->y,theta0,in_pose->theta);


/* Initialize world database for fifth floor of Spanagle Hall, determine
2D view of environment, and label the vertical model features. */

 FifthFloor = make_world();
 ModelLinesHead1 = get_view(in_pose->x, in_pose->y, CAMERA_HEIGHT, theta0,
                                    FifthFloor,FOCAL_LENGTH);

 printf("Nr Vertical Model Lines = %d\n",ModelLinesHead1->VERT_LINES);
 label_model_lines(ModelLinesHead1->VLINE_LIST, in_pose);


 /* Call to update_pose() */
 corr_pose = update_pose(ImageLinesHead,ModelLinesHead1->VLINE_LIST,in_pose);

theta0 = corr_pose->theta * 180.0 / PI;
 if(theta0 < 0.0) theta0 += 360.0;

 printf("\nCorrected pose: x= %.2f, y= %.2f, theta= %.2f(%.4f rads)\n",
                corr_pose->x,corr_pose->y,theta0,corr_pose->theta);

/* Get 2D view of world from corrected pose. */
 ModelLinesHead2 = get_view(corr_pose->x, corr_pose->y, CAMERA_HEIGHT, theta0,
                                         FifthFloor, FOCAL_LENGTH);

/* Display image, 2D view from input pose, and 2D view from corrected pose. */
 display_match_image(img, ImageLinesHead, ModelLinesHead1, ModelLinesHead2);


 free_lines(ModelLinesHead1);
 free_lines(ModelLinesHead2);

 printf(" vertmatch %s...done.\n",argv[1]);
 }
```

B. FILE: MATCHSUPPORT.H

```
/**********************************************************************/
/* FILENAME: matchsupport.h
/* AUTHOR: Kevin Peterson
/* DATE: 09 March 1992
/**********************************************************************/
#define FOCAL_LENGTH 1.40 /* cm */
#define FOCAL_LENGTH_IN_PIXELS 1205.4524 /* number of pixels */
#define VERT_PICTURE_BORDER 8.0 /* number of pixels */

#define FIELD_HALF_ANGLE 0.2556 /* radians = 14.645 degrees */

#define CAMERA_HEIGHT 40.0 /* inches */

#define MIN_CHORD_LENGTH 10.0 /* min dist between 2 vertical
                                 model features (in inches)
                                 used in determine_position()*/

#define DELTA_LENGTH_RATIO 0.05 /* Endpoint delta/IMG_LINE length
                                   used in determine_vert_match()*/

#define IMG_LINE_MIN_ANGLE 0.0070 /* radians = 0.20 degrees */

#define VERIFY_EQ_ANGLE_EPSILON 0.0035 /* radians = 0.20 degrees */

/*---------------------------------------------------------------*/
/* double normalize(doble alpha)
/*
/* Returns an angle between pi and -pi.
/*---------------------------------------------------------------*/
double normalize(alpha)
        double alpha;
{
        while(alpha > PI) alpha -= (2.0*PI);
        while(alpha < -PI) alpha += (2.0*PI);
        return alpha;
}

/*---------------------------------------------------------------*/
/*
 Returns 1 if angle a1 is "left_of" angle a2.
/*---------------------------------------------------------------*/
*/
int left_of(a1, a2)
        double a1, a2;
{
        return (normalize(a1 - a2) > 0.0);
}

/*---------------------------------------------------------------*/
/*
 Returns angle for use with environment model coordinates.
 Environment model uses angles measured from y-axis.
/*---------------------------------------------------------------*/
*/
double map_angle(alpha)
        double alpha;
{
        return normalize(alpha - (PI/2.0)); /* alpha_prime */
}
```

```
/*--------------------------------------------------------------*/
/* void label_model_lines(LINE *m, POSE *p)
/*
/* Routine to determine lengths, orientations, angles from image center, and
/* labels for all model LINEs visible in image.
/*--------------------------------------------------------------*/

label_model_lines(m, p)
            LINE *m;
            POSE *p; /* estimated camera POSE */
{

 unsigned char first_char = ' ', second_char = 'A';


 /* Note: Vertical edge extraction produces IMG_LINEs with p1.y always
 less than p2.y. Similarly for vertical model LINEs, Y1 is always
 less than Y2.
 */

 printf("\nModel Lines: ------\n");

 while(m != NULL)
            {

                m->length = m->Y2 - m->Y1;
                m->est_pose_orient = map_angle(atan2(m->MODEL_Y - p->y,
                                        m->MODEL_X - p->x));
                m->est_angle_to_image_center = m->est_pose_orient - p->theta;

                if(m->length > 15.0) /* temporary -- don't do very small lines */
                {
                    strcpy(m->name," ");
                    sprintf(m->name, "%c%c", first_char, second_char);

                    if(second_char == 'Z')
                        {
                            if(first_char == ' ') first_char = 'A';
                            else first_char = first_char + 1;
                            second_char = 'A';
                        }
                    else second_char = second_char + 1;

                    printf(" %s: (%.1f,%.1f) length= %.2f, psi= %.2f, psi-theta0= %.4f\n",
                            m->name, m->MODEL_X, m->MODEL_Y, m->length,
                            m->est_pose_orient, m->est_angle_to_image_center);
                }
                else
                {
                    sprintf(m->name, " ");
                }

                m = m->NEXT;

            }

}
```

```
/*--------------------------------------------------------------*/
/* void determine_vert_match(LINE *m, IMG_LINE *i)
/*
/* Function to determine confidence value for a possible match
/* based upon translation, rotation, scaling, and overlapping
/* of endpoints between a LINE from the model list and an
/* IMG_LINE from the image list.
/*--------------------------------------------------------------*/

determine_vert_match(m,i)
        LINE *m; /* ptr to LINE from model list */
        IMG_LINE *i; /* ptr to IMG_LINE from image list */
{
 int endpoint_inclusion;
 double delta;
 MATCHTYPE *match, *place;


/* Endpoint_Inclusion of image endpoints: for vertical image lines, i->p1 is
always lower than i->p2. */

delta = (m->Y1 - i->p1.y) + (i->p2.y - m->Y2);

if((delta <= 0.0) || ((delta > 0.0) && (delta/i->dmajor < DELTA_LENGTH_RATIO)))
 {
        /* then create MATCHTYPE and place in the IMG_LINE's matchlist */

        if((match = (MATCHTYPE *)malloc(sizeof(MATCHTYPE))) == NULL)
           fatal("determine_match: creating MATCHTYPE, malloc\n");

        match->line = m;
        match->angle_view_diff = (m->est_angle_to_image_center - i->angle_to_image_center);
        match->next = NULL;

        /* Place match in matchlist by increasing order of angle_view_diff. */

        if (i->matchlist == NULL) i->matchlist = match;
        else if(fabs(match->angle_view_diff) < fabs(i->matchlist->angle_view_diff))
           {
              match->next = i->matchlist;
              i->matchlist = match;
           }
        else
           {
              place = i->matchlist;
              while((place->next != NULL) && (fabs(match->angle_view_diff) >
                                                fabs(place->next->angle_view_diff)))
                 {
                    place = place->next;
                 }
              match->next = place->next;
              place->next = match;
           }

 } /* end if(delta...) */

}
```

```
/*---------------------------------------------------------------*/
/* void find_vertical_matches(IMG_LINE *imgline, LINE *modelline)
/*
/* Function to find all possible vertical model LINEs that could be matched
/* to a vertical IMG_LINE.
/*---------------------------------------------------------------*/
find_vertical_matches(imgline,modelline)
            IMG_LINE *imgline;
            LINE *modelline;
{

 while (modelline != NULL)
            {
                determine_vert_match(modelline,imgline);
                modelline = modelline->NEXT;
            }

 imgline->pm = imgline->matchlist;

}

/*---------------------------------------------------------------*/
/* void print_matchlist(IMG_LINE *l)
/*
/* Function to output the model LINEs matched to an IMG_LINE.
/*-------------------- ------------------------------------------*/
print_matchlist(l)
            IMG_LINE *l;
{

 MATCHTYPE *m = l->matchlist;

 while(m != NULL)

                printf(" %s(%.4f)", m->line->name, m->angle_view_diff);
                        m = m->next;


 printf("\n\n");

}

/*---------------------------------------------------------------*/
/* void declare_matches(IMG_LINE *l, IMG_LINE *m, IMG_LINE *r,
/*                                     LINE *modellines)
/*
/* Declare all matches to model LINEs for left, middle, and right
/* vertical IMG_LINEs.
/*---------------------------------------------------------------*/
declare_matches(l,m,r,modellines)
            IMG_LINE *l, *m, *r;
            LINE *modellines;
{
 find_vertical_matches(l,modellines);
 find_vertical_matches(m,modellines);
 find_vertical_matches(r,modellines);
 printf("\nImage line matchlists to Model line NAME(angle difference): -----\n");
 printf("left (edge %s) > \n",l->name); print_matchlist(l);
 printf("middle (edge %s) > \n",m->name); print_matchlist(m);
 printf("right (edge %s) > \n",r->name); print_matchlist(r);
}
```

```
/*---------------------------------------------------------------*/
/* int determine_postiont(LINE *l, LINE *m, LINE *r, double alpha1, double alpha2,
/*                         POSE *p, double *x, double *y)
/*
/* Routine to determine correct position based upon 3 matched model LINEs and
/* the measured angle differences from the IMG_LINEs. Returns 1 if x,y
/* position is determined, 0 otherwise.
/*---------------------------------------------------------------*/

int determine_position(l, m, r, alpha1, alpha2, p, x, y)
        LINE *l, *m, *r; /* left, middle, and right viewed model LINEs */
        double alpha1, alpha2; /* measured image angles */
        POSE *p; /* estimated (input) POSE */
        double *x, *y; /* x,y location to be returned */
{
double chord1_length, chord2_length,
            chord1_mid_x, chord1_mid_y, chord2_mid_x, chord2_mid_y,
            chord1_orient, chord2_orient,
            h1_orient, h2_orient, h1, h2,
            pv1_x, pv1_y, pv2_x, pv2_y,
            circle1_radius, circle2_radius,
            circle1_x, circle1_y, circle2_x, circle2_y,
            c2c1, c2c1_orient, c2robot_orient, c2_omega;

/* Ensure that 3 different model LINEs are begin used. */
if((l == m) || (m == r) || (l == r))
{
        printf("NO position: 2 IMG_LINEs matched to same model LINE\n");
        return 0;
}

/* Determine chord lengths, midpoints, and orientations. */
chord1_length =
        sqrt((l->MODEL_X - m->MODEL_X )*(l->MODEL_X - m->MODEL_X )) +
            ((l->MODEL_Y - m->MODEL_Y )*(l->MODEL_Y - m->MODEL_Y )));

chord2_length =
        sqrt((m->MODEL_X - r->MODEL_X )*(m->MODEL_X - r->MODEL_X )) +
            ((m->MODEL_Y - r->MODEL_Y )*(m->MODEL_Y - r->MODEL_Y )));

if((chord1_length < MIN_CHORD_LENGTH) || (chord2_length < MIN_CHORD_LENGTH))
{
        printf("NO position: chord length < MIN_CHORD_LENGTH\n");
        return 0;
}

chord1_mid_x = (l->MODEL_X + m->MODEL_X )/2.0;
chord1_mid_y = (l->MODEL_Y + m->MODEL_Y )/2.0;
chord2_mid_x = (m->MODEL_X + r->MODEL_X )/2.0;
chord2_mid_y = (m->MODEL_Y + r->MODEL_Y )/2.0;

chord1_orient = map_angle(atan2(l->MODEL_Y - m->MODEL_Y,
                                l->MODEL_X - m->MODEL_X));
chord2_orient = map_angle(atan2(m->MODEL_Y - r->MODEL_Y,
                                m->MODEL_X - r->MODEL_X));

/* Determine orientations perpendicular to the circle chords towards input pose. */
h1_orient = normalize(chord1_orient + (PI/2.0));
h2_orient = normalize(chord2_orient + (PI/2.0));
```

```
/* Determine perpendicular lengths from the center of the chords to a point on the viewing circle. */
h1 = (chord1_length/2.0) / tan(alpha1/2.0);
h2 = (chord2_length/2.0) / tan(alpha2/2.0);

/* Determine the viewing points perpendicular to the chords. */
pv1_x = chord1_mid_x - (h1 * sin(h1_orient));
pv1_y = chord1_mid_y + (h1 * cos(h1_orient));
pv2_x = chord2_mid_x - (h2 * sin(h2_orient));
pv2_y = chord2_mid_y + (h2 * cos(h2_orient));

/* Determine radii and centers of viewing circles. */
circle1_radius = sqrt((h1*h1) + (chord1_length*chord1_length/4.0))/2.0;
circle2_radius = sqrt((h2*h2) + (chord2_length*chord2_length/4.0))/2.0;

circle1_x = pv1_x - (circle1_radius * sin(h1_orient + PI));
circle1_y = pv1_y + (circle1_radius * cos(h1_orient + PI));
circle2_x = pv2_x - (circle2_radius * sin(h2_orient + PI));
circle2_y = pv2_y + (circle2_radius * cos(h2_orient + PI));

/* Determine distance and orientation between centers of viewing circles. */
c2c1 = sqrt(((circle1_x - circle2_x)*(circle1_x - circle2_x)) +
            ((circle1_y - circle2_y)*(circle1_y - circle2_y)));

if(c2c1 == 0.0)
{
        printf("NO position: viewpoint circles identical\n");
        return 0;
}

c2c1_orient = map_angle(atan2(circle1_y - circle2_y, circle1_x - circle2_x));
c2robot_orient = map_angle(atan2(p->y - circle2_y, p->x - circle2_x));

/* Calculate viewing position. */

if(circle2_radius < fabs(c2c1 - circle1_radius)) c2_omega = 0.0;
else if(circle2_radius > (c2c1 + circle1_radius)) c2_omega = PI;
else c2_omega = acos((c2c1 * c2c1) + (circle2_radius * circle2_radius) -
                (circle1_radius * circle1_radius)) / (2 * c2c1 * circle2_radius));

if(left_of(c2robot_orient, c2c1_orient))
{
        *x = circle2_x - (circle2_radius * sin(c2c1_orient+c2_omega));
        *y = circle2_y + (circle2_radius * cos(c2c1_orient+c2_omega));
}
else
{
        *x = circle2_x - (circle2_radius * sin(c2c1_orient-c2_omega));
        *y = circle2_y + (circle2_radius * cos(c2c1_orient-c2_omega));
}

return 1;

}
```

```
/*---------------------------------------------------------------*/
/* void select_next_best_match (MATCHTYPE *l, MATCHTYPE *m,
/*                                    MATCHTYPE *r, int *k)
/*
/* Select next best model LINE (minimum fabs(angle_view_diff)) for new
/* combination in the next determine_position() attempt.
/*---------------------------------------------------------------*/

select_next_best_match(l,m,r,k)
        MATCHTYPE *l, *m, *r; /* MATCHTYPE pointers for left, middle and right
                                    IMG_LINES */
        int *k;                  /* number of possible pm->next != NULL */
{
switch(*k)
 {
 case 3:
        if((fabs(l->next->angle_view_diff) < fabs(m->next->angle_view_diff))
            && (fabs(l->next->angle_view_diff) < fabs(r->next->angle_view_diff)))
            *k = 1;
        else if((fabs(m->next->angle_view_diff) < fabs(l->next->angle_view_diff))
            && (fabs(m->next->angle_view_diff) < fabs(r->next->angle_view_diff)))
            *k = 2;
        else *k = 3;
        break;

 case 2:
        if(l->next == NULL)
            {
                if(fabs(m->next->angle_view_diff) < fabs(r->next->angle_view_diff)) *k = 2;
                else *k = 3;
            }
        else if(m->next == NULL)
            {
                if(fabs(l->next->angle_view_diff) < fabs(r->next->angle_view_diff)) *k = 1;
                else *k = 3;
            }
        else
            {
                if(fabs(l->next->angle_view_diff) < fabs(m->next->angle_view_diff)) *k = 1;
                else *k = 2;
            }
        break;

 case 1:
        if(l->next != NULL) *k = 1;
        else if(m->next != NULL) *k = 2;
        else *k = 3;
        break;

 default:
        *k = 0;
        break;

 } /* end switch */

}
```

```
/*--------------------------------------------------------------------*/
/* double determine_rotation (LINE *l, LINE *m, LINE *r, double x, double y,
/*                                        double intheta, double loffang, double moffang,
/*                                        double roffang)
/*
/* Function that determines the average amount of rotational correction
/* that is required based upon the 3 model LINEs angular offset from the
/* IMG_LINEs to which they are matched with.
/*--------------------------------------------------------------------*/

double determine_rotation(l, m, r, x, y, intheta, loffang, moffang, roffang)
        LINE *l, *m, *r;
        double x, y, intheta, loffang, moffang, roffang;
{

/* calculate orientations of new position to Vertical Model LINEs */

double lneworient = map_angle(atan2(l->MODEL_Y - y, l->MODEL_X - x)),
mneworient = map_angle(atan2(m->MODEL_Y - y, m->MODEL_X - x)),
rneworient = map_angle(atan2(r->MODEL_Y - y, r->MODEL_X - x)),


/* calculate angles from center of image for new position */

lnewestangle = normalize(lneworient - intheta),
mnewestangle = normalize(mneworient - intheta),
rnewestangle = normalize(rneworient - intheta),


/* calculate differences with offset angles from image */

l_diff = normalize(lnewestangle - loffang),
m_diff = normalize(mnewestangle - moffang),
r_diff = normalize(rnewestangle - roffang),


/* average rotation difference */

rotation_diff = (l_diff + m_diff + r_diff) / 3.0;


return rotation_diff;

}
```

```
/*--------------------------------------------------------------*/
/* int verify_pose(double x, double y, double theta, IMG_LINE *ImgLines,
/*                         LINE *ModelLines)
/*
/* Function to return number of IMG_LINEs that lie over model LINEs
/* for a corrected 2D view of the model environment for a given possible pose.
/*--------------------------------------------------------------*/
int verify_pose(x,y,theta,ImgLines,ModelLines)
        double x,y,theta;
        IMG_LINE *ImgLines;
        LINE *ModelLines;
{
LINE            *ml = ModelLines;
IMG_LINE *il = ImgLines;
double          delta;
int             endpoint_inclusion = 0, imgline_on_modelline = 0, nr_good_hits = 0,
                found = 0;

/* Calculate est_angle_to_image_center for all ModelLines based upon
 pose: (x,y,theta). */
 while(ml != NULL)
 {
        ml->est_angle_to_image_center =
           normalize(map_angle(atan2(ml->MODEL_Y-y, ml->MODEL_X-x)) - theta);
        ml = ml->NEXT;
 }

 /* Find correspondences of image lines to model lines. */
while(il != NULL)
 {
        /* Ensure vertical image line is not due to image borders. */
        if(fabs(il->angle_to_image_center) < FIELD_HALF_ANGLE)
            {
                /* Find a corresponding IMG_LINE. */
                ml = ModelLines;
                found = 0;
                while((ml != NULL) && !found)
                {
                    delta = (ml->Y1 - il->p1.y) + (il->p2.y - ml->Y2);

                    endpoint_inclusion =
                        ((delta <= 0.0) ||
                         ((delta > 0.0) && (delta/il->dmajor < DELTA_LENGTH_RATIO)));

                    imgline_on_modelline =
                        (fabs(il->angle_to_image_center -
                         ml->est_angle_to_image_center)<VERIFY_EQ_ANGLE_EPSILON);

                    if(endpoint_inclusion && imgline_on_modelline)
                        {
                            ++nr_good_hits;
                            found = 1;
                        }

                    ml = ml->NEXT;
                }
            }
        il = il->next;
 }
return nr_good_hits;
}
```

```
/*-------------------------------------------------------------------*/
/* POSE *calculate_best_pose(IMG_LINE *l, IMG_LINE *m, IMG_LINE *r,
/*                          double alpha_lm, double alpha_mr, POSE *in_pose,
/*                          IMG_LINE *ImgLines, LINE *ModelLines)
/*
/* Function that conducts several iterations of finding an IMG_LINE to LINE
/* triplet, determines the POSE based upon the matching, and returns the best POSE
/* based upon number of IMG_LINE to model LINE matches of a 2D view of the
/* best POSE and translational error of the robot.
/*-------------------------------------------------------------------*/

POSE *calculate_best_pose(l,m,r,alpha_lm,alpha_mr,in_pose,ImgLines,ModelLines)
        IMG_LINE *l, *m, *r, *ImgLines;
        double alpha_lm, alpha_mr;
        POSE *in_pose;
        LINE *ModelLines;
{
 int              j = 0, k = 0, nr_hits = 0, best_nr_hits = 0;
 POSE             *best_pose;
 double           x, y, rot, trans = 0.0, best_rot, best_trans;
 MATCHTYPE        *lmatch = l->matchlist,
                  *mmatch = m->matchlist,
                  *rmatch = r->matchlist;


 printf("Pose determination: -----\n");

 if((best_pose = (POSE *)malloc(sizeof(POSE))) == NULL)
        fatal("calculate_best_pose: malloc\n");


 if((lmatch == NULL) || (mmatch == NULL) ||(rmatch == NULL))
        {
        printf("Can not determine pose for %s, %s, %s (NULL matchlist)\n",
                l->name,m->name,r->name);
        }

else for(j = 0; j < 20; ++j)
        {

        printf(" [%s,%s,%s] ",lmatch->line->name,mmatch->line->name,
                rmatch->line->name);

        if(determine_position(lmatch->line, mmatch->line, rmatch->line,
                          alpha_lm, alpha_mr, in_pose, &x, &y))
           {

                /* Calculate translation and rotation differences. */

                trans = sqrt((in_pose->x - x) * (in_pose->x - x)) +
                          ((in_pose->y - y) * (in_pose->y - y));

                rot = determine_rotation(lmatch->line, mmatch->line, rmatch->line,
                             x, y, in_pose->theta,
                             l->angle_to_image_center,
                             m->angle_to_image_center,
                             r->angle_to_image_center);

                nr_hits = verify_pose(x,y,in_pose->theta + rot,ImgLines,ModelLines);
```

```
            printf("x= %.2f y= %.2f T= %.4f R= %.4f nr verify matches= %d\n",
                x,y,trans,rot,nr_hits);

            if((j == 0) || (((double)nr_hits)/trans > ((double)best_nr_hits)/best_trans))
            {
                best_nr_hits = nr_hits; /* save best trans and rot values */
                best_trans = trans;
                best_rot = rot;
                best_pose->x = x; /* assign values to best_pose */
                best_pose->y = y;
                best_pose->theta = in_pose->theta + rot;
                printf(" *** BEST POSE MODIFIED ***\n");
                l->pm = lmatch; /* set 3 IMG_LINE's match pointers */
                m->pm = mmatch;
                r->pm = rmatch;
            }

        }

    /* Select the next best match combination of model LINEs. */

    k = ((lmatch->next != NULL) + (mmatch->next != NULL) +
        (rmatch->next != NULL));

    if(k > 0)
        {
            select_next_best_match(lmatch,mmatch,rmatch,&k);
            if(k == 1) lmatch = lmatch->next;
            else if(k == 2) mmatch = mmatch->next;
            else if(k == 3) rmatch = rmatch->next;
        }
    else return best_pose;

}

return best_pose;

}
```

```
/*--------------------------------------------------------------*/
/* POSE *update_pose(IMG_LINE *ImageLines, LINE *ModelLines, POSE *in_pose)
/*
/* Function to return a corrected POSE given the vertical edges extracted from an image
/* and the vertical model LINEs that should be visible from the expected POSE of the
/* robot.
/*--------------------------------------------------------------*/

POSE *update_pose(ImageLines,ModelLines,in_pose)
        IMG_LINE *ImageLines;
        LINE *ModelLines;
        POSE *in_pose;
{
IMG_LINE *il, *left, *middle, *right;
POSE            *out_pose;
double          alpha_lm, alpha_mr;
int             j;


if((out_pose = (POSE *)malloc(sizeof(POSE))) == NULL)
        fatal("update_pose: creating POSE out_pose: malloc\n");


/* Find the 3 most significant IMG_LINEs and return them in
left-to-right order pointed to by *left, *middle, and *right. */

il = ImageLines;
printf("\nDetermine 3 major image edges for matching: -----\n");


for(j = 1; j < 4;)
{
        if(il == NULL) fatal("CAN NOT DETERMINE POSE: < 3 image lines\n");

        /* Ensure vertical image line is not due to image borders. */

        if(fabs(il->angle_to_image_center) < FIELD_HALF_ANGLE)
            {

                if(j == 1)
                {
                    left = il;
                    printf(" %d) line %s, angle from center = %.4f\n", j,
                            il->name, il->angle_to_image_center);
                    ++j;
                }

                else if((j == 2) && (fabs(il->angle_to_image_center -
                            left->angle_to_image_center) > IMG_LINE_MIN_ANGLE))
                {
                    if(il->angle_to_image_center > left->angle_to_image_center)

                        right = left; left = il;
                    }
                   else right = il;

                    printf(" %d) line %s, angle from center = %.4f\n", j,
                        il->name, il->angle_to_image_center);

                    ++j;
                }
```

```
                else if((j ==3) &&
                    (fabs(il->angle_to_image_center -
                        left->angle_to_image_center) > IMG_LINE_MIN_ANGLE) &&
                    (fabs(il->angle_to_image_center -
                        right->angle_to_image_center) > IMG_LINE_MIN_ANGLE))
                {
                    if(il->angle_to_image_center > left->angle_to_image_center)
                        {
                        middle = left; left = il;
                        }

                    else if(il->angle_to_image_center < right->angle_to_image_center)
                        {
                        middle = right; right = il;
                        }

                else middle = il;

                printf("%d) line %s, angle from center = %.4f\n", j,
                        il->name, il->angle_to_image_center);

                ++j;

                }
            }

        il = il->next;
    }


/* Calculate two angles between the three major IMG_LINEs. */

alpha_lm = left->angle_to_image_center - middle->angle_to_image_center;
alpha_mr = middle->angle_to_image_center - right->angle_to_image_center;


/* Declare the matches to model LINEs for the three major IMG_LINEs. */

declare_matches(left,middle,right,ModelLines);


/* Calculate the correct pose. */

out_pose = calculate_best_pose(left, middle, right, alpha_lm, alpha_mr,
                                in_pose, ImageLines, ModelLines);

return out_pose;

}
```

C. FILE: MATCHDISPLAYSUPPORT.H

```
/**********************************************************************/
/* FILENAME: matchdisplaysupport.h
/* AUTHOR: Kevin Peterson
/* DATE: 07 February 1992
/*
/*DESCRIPTION: Collection of display functions.
/*
/* void display_match_image (NPSIMAGE *img, IMG_LINE *l,
/*                              LINE_HEAD *m1, LINE_HEAD *m2)
/* void draw_white_lines(IMG_LINE *l)
/* void draw_white_model_lines_with_names (LINE *m)
/* void draw_white_model_lines (LINE *m)
/* void draw_black_model_lines_with_names (LINE *m)
/* void draw_black_model_lines (LINE *m)
/* void draw_match_lines (IMG_LINE *l)
/* void display_match_loop (NPSIMAGE *img, IMG_LINE *l,
/*                              LINE_HEAD *m1, *LINE_HEAD *m2, long winid)
/*
/* These functions make calls to following SiliconGraphics routines:
/* prefsize(), winopen(), winset(), winclose(), RGBmode(),
/* singlebuffer(), gconfig(), qdevice(), c3f(), move2(), draw2(),
/* swapbuffers(), lrectwrite(), reshapeviewport(), cmov2(),
/* charstr(), and clear().
/**********************************************************************/


/*------------------------------------------------------------*/
/* void display_match_image (NPSIMAGE *img, IMG_LINE *l,
/* LINE_HEAD *m1, LINE_HEAD *m2)
/*------------------------------------------------------------*/
display_match_image(img,l,m1,m2)
          NPSIMAGE *img; /* input image */
          IMG_LINE *l; /* list of IMG_LINEs found from image */
          LINE_HEAD *m1, m2; /* LINE_HEAD types from Jim's "graphics.c" */
{

 long winid;


 prefsize(img->xsize,img->ysize); /* preferred size for window */
 winid = winopen(img->name); /* open the window */
 RGBmode(); /* set RGBmode, singlebuffer, and */
 singlebuffer(); /* configure the window */
 gconfig();

 /* initialize_controls */
 qdevice(REDRAW);
 qdevice(LEFTMOUSE);
 qdevice(MIDDLEMOUSE);
 qdevice(RIGHTMOUSE);
 qdevice(ESCKEY);

 display_match_loop(img,l,m1,m2,winid);

 free(img->imgdata.bitsptr); /* delete the bitmap for the image */
 free(img); /* delete the NPSIMAGE structure */
 winclose(winid); /* close the window */
}
```

```
/*-------------------------------------------------------------*/
/* void draw_white_lines(IMG_LINE *l)
/*
/* Draws white lines over an image.
/*-------------------------------------------------------------*/
draw_white_lines(l)
            IMG_LINE *l;
{
 static float white[3] = {1.0,1.0,1.0}; /* rgb white */

 c3f(white);
 while(l!=NULL)
 {
            cmov2(l->p1.x,l->p1.y); charstr(l->name);
            move2(l->p1.x,l->p1.y); draw2(l->p2.x,l->p2.y);
            l = l->next;
 }
 swapbuffers();
}


/*-------------------------------------------------------------*/
/* void draw_white_model_lines_with_names (LINE *m)
/*
/* Draws blue model lines over an image.
/*-------------------------------------------------------------*/
draw_white_model_lines_with_names(m)
            LINE *m;
{
 static float white[3] = {1.0,1.0,1.0}; /* rgb white */

 c3f(white);
 while(m!=NULL)
 {
            cmov2(m->X1,m->Y1); charstr(m->name);
            move2(m->X1,m->Y1); draw2(m->X2,m->Y2);
            m = m->NEXT;
 }
 swapbuffers();
}


/*-------------------------------------------------------------*/
/* void draw_black_model_lines_with_names (LINE *m)
/*
/* Draws black model lines over an image.
/*-------------------------------------------------------------*/
draw_black_model_lines_with_names(m)
            LINE *m;
{
 static float black[3] = {0.0,0.0,0.0}; /* rgb black */

 c3f(black);
 while(m!=NULL)
 {
            cmov2(m->X1,m->Y1); charstr(m->name);
            move2(m->X1,m->Y1); draw2(m->X2,m->Y2);
            m = m->NEXT;
 }
 swapbuffers();
}
```

```
/*------------------------------------------------------------------*/
/* void draw_white_model_lines (LINE *m)
/*
/* Draws white model lines over an image.
/*------------------------------------------------------------------*/
draw_white_model_lines(m)
            LINE *m;
{
 static float white[3] = {1.0,1.0,1.0}; /* rgb white */

 c3f(white);
 while(m!=NULL)
 {
            move2(m->X1,m->Y1); draw2(m->X2,m->Y2);
            m = m->NEXT;
 }
 swapbuffers();
}
```

```
/*------------------------------------------------------------------*/
/* void draw_black_model_lines (LINE *m)
/*
/* Draws black model lines over an image.
/*------------------------------------------------------------------*/
draw_black_model_lines(m)
            LINE *m;
{
 static float black[3] = {0.0,0.0,0.0}; /* rgb black */

 c3f(black);
 while(m!=NULL)
 {
            move2(m->X1,m->Y1); draw2(m->X2,m->Y2);
            m = m->NEXT;
 }
 swapbuffers();
}
```

```
/*-----------------------------------------------------------------*/
/* void display_match_loop (NPSIMAGE *img, IMG_LINE *l,
/* LINE_HEAD *m1, LINE_HEAD *m2, long winid)
/*-----------------------------------------------------------------*/
display_match_loop(img,l,m1,m2,winid)
        NPSIMAGE *img; /* input image */
        IMG_LINE *l; /* list of IMG_LINEs found from image */
        LINE_HEAD *m1, *m2; /* LINE_HEAD type from Jim's "graphics.c" */
        long winid;
{
static float white[3] = {1.0,1.0,1.0}; /* rgb white */
short value; /* value returned from the event queue */


/* display the images once */

winset(winid);
lrectwrite(0,0,img->xsize - 1,img->ysize - 1,img->imgdata.bitsptr);
draw_white_lines(l);


/* loop until a mouse button is pressed */
while(TRUE)
{

switch(qread(&value))
        {
        case REDRAW:
            winset((long)value);
            reshapeviewport();
            if(value == winid)
            {
                lrectwrite(0,0,img->xsize - 1,img->ysize - 1, img->imgdata.bitsptr);
                draw_white_lines(l);
            }
            break;

        case LEFTMOUSE:
            if(value == 0)
            {
                lrectwrite(0,0,img->xsize - 1,img->ysize - 1, img->imgdata.bitsptr);
                draw_white_model_lines_with_names(m1->VLINE_LIST);
                draw_white_model_lines(m1->LINE_LIST);
            }
            break;

        case MIDDLEMOUSE:
            if(value == 0)
            {
                c3f(white);
                clear();
                draw_black_model_lines_with_names(m1->VLINE_LIST);
                draw_black_model_lines(m1->LINE_LIST);
            }
            break;

        case RIGHTMOUSE:
            if(value == 0)
            {
                lrectwrite(0,0,img->xsize - 1,img->ysize - 1, img->imgdata.bitsptr);
                draw_white_model_lines(m2->VLINE_LIST);
```

```
                draw_white_model_lines(m2->LINE_LIST);
            }
            break;

        case ESCKEY:
            return;

        default:
            break;

        } /* end switch */

    } /* end while */

}
```

APPENDIX D - USER'S GUIDE

```
file: README
date: 8 March 92
by : Kevin Peterson

USER'S MANUAL FOR YAMABICO'S VISUAL NAVIGATION ROUTINES

 This file outlines the functions of the following programs:
     findedge, fastedge, and vertmatch

1. REQUIREMENTS:

  1.1 HARDWARE: The above routines were developed on a
SiliconGraphics Personal Iris workstation and can be run on
all SGI stations at NPS.

  1.2 SOFTWARE: Ensure the following files are in the
local directory for the edge extraction routines:
 README (this file)
 Makefile
 image_types.h
 match_types.h
 npsimagesupport.h
 edgesupport.h
 vertsupport.h
 displaysupport.h
 findedge.c
 fastedge.c

Ensure the following files are in the local directory for the
matching   pose determination routines:
 5th.h
 2d-.h
 graphics.h
 matchdisplaysupport.h
 matchsupport.h
 vertmatch.c


2. COMPILATION:

 2.1 The following C program libraries are required:
 device.h
 math.h
 stdio.h
 gl.h
 gl/image.h

 2.2 findedge, casedge, fastedge, fastsobel, vertedge,
and vertmatch can all be compiled by the command line similar to:
 cc -o findedge findedge.c -limage -lgl_s -lm
 cc -o fastedge fastedge.c -limage -lgl_s -lm
 etc...

or if the Makefile is present:
 make findedge
 make fastedge
 etc...
```

3. PROGRAM FUNCTIONS:

3.4 FINDEDGE - Displays the grayscale and gradient images of the input image. Conducts edge extraction based upon least squares fit.

3.6 FASTEDGE - Displays the input image and edges extracted. Uses the green component of the RGB pixel value vice the grayscale value and does not create the grayscale or gradient images.

3.9 VERTMATCH - Extracts vertical edges from the input image (VERTEDGE), creates a 2D view of the model environment based upon the estimated pose (dead-reckoning maintained configuration) of the robot, conducts line matching between image vertical edges and model vertical line segments, determines possible poses based upon different matchings for the three major image edges, then returns the best pose based upon number of edge to model line matches and distance of updated pose to the estimated (input) pose.

4. USING THE PROGRAMS:

4.4 findedge <filename>
Filename is input picture file to extract edges from. Two windows will be created; one for the grayscale image, the other for the gradient image.
- Lines extracted can be drawn onto either image by the LEFT or MIDDLE mouse buttons while the cursor is in the TITLE_BAR.
- Lines can be drawn over a white background by MIDDLE mouse button while the cursor is in the WINDOW.
- Exit: RIGHT or LEFT mouse buttons (cursor in WINDOW) or escape key.

4.6 fastedge <filename>
One window is opened to display the input (color) image with extracted edges drawn over the image.
- Lines extracted can be drawn onto either image by the LEFT mouse buttons while the cursor is in the WINDOW.
- Lines can be drawn over a white background by MIDDLE mouse button while the cursor is in the WINDOW.
- Exit: LEFT mouse buttons (cursor in WINDOW) or escape key.

4.9 vertmatch <filename>
One window is opened to display the input image with the extracted vertical edges.
- MIDDLE mouse button (while in WINDOW) displays the 2D wire-frame view for the expected pose over a white background.
- RIGHT mouse button displays 2D expected pose wire-frame view over the input picture.
- LEFT mouse button displays 2D wire-frame view from the corrected pose over the input picture.
- Escape key exits.

REFERENCES

[BAL82] Ballard, D.H., *Computer Vision*, Prentice-Hall, 1982.

[BEV90] Beveridge, J.R., Weiss, R., and Riseman, E.M., "Combinatorial Optimization Applied to Variable Scale 2D Model Matching," *1990 IEEE 10th International Conference on Pattern Recognition*, vol. 1, pp. 18-23, June 1990.

[FEN90] Fennema, C., and Hanson, A.R., "Experiments in Autonomous Navigation," *1990 IEEE 10th International Conference on Pattern Recognition*, vol. 1, pp. 24-31, June 1990.

[HAL89] Hall, R., *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, 1989.

[HAR89] Haralick, R.M., Joo, H., Lee, C., Zhuang, X., Vaidya, V.G., and Kim, M., "Pose Estimation from Corresponding Point Data," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 19, no. 6, pp. 1427-1446, November/December 1989.

[HEL89] Heller, A.J., and Stenstrom, J.R. "Verification of Recognition and Alignment Hypotheses by Means of Edge Verification Statistics," *Proc. DARPA Image Understanding Workshop*, pp. 957-966, May 1989.

[MAR79] Marr, D., "Representing and Computing Visual Information," *Artificial Intelligence: An MIT Perspective*, vol. 2, 1979.

[MOF80] Moffit, F.H., and Mikhail, E.M., *Photogrammetry* (3rd ed.), Harper & Row, 1980.

[KAN89a] Kanayama, Y., and Onishi, M., "Locomotion Functions for a Mobile Robot Language," *Proc. IEEE Workshop on Intelligent Robots and Systems*, pp. 542-549, September 1989.

[KAN89b] Technical Report of the Department of Computer Science , University of California at Santa Barbara, Report TRCS89-06, *Spatial Learning by an Autonomous Mobile Robot with Ultrasonic Sensors*, by Kanayama, Y., and Noguchi, T., February 1989.

[KUM89] Kumar, R., "Determination of Camera Location and Orientation," *Proc. DARPA Image Understanding Workshop*, pp. 870-887, May 1989.

[LIU90] Liu, Y., Huang, T.S., and Faugeras, O.D., "Determination of Camera Location from 2D to 3D Line and Point Correspondences," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 1, pp. 28-37, January 1990.

[NEV82] Nevatia, R., *Machine Perception*, Prentice-Hall, 1982.

[SHE91] Sherfey, S., *A Mobile Robot Sonar System*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

[STE92] Stein, J., *Modelling Visibility Testing and Projection of an Orthogonal Three Dimensional World in Support of a Single Camera Vision System*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1992.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria VA 22304-6145	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey CA 93943-5002	2
Dr. Robert B. McGhee Code CS/Mz Chairman, Computer Science Department Naval Postgraduate School Monterey CA 93943-5000	1
Dr. Yutaka Kanayama Code CS/Ka Computer Science Department Naval Postgraduate School Monterey CA 93943-5000	2
Dr. Neil C. Rowe Code CS/Rp Computer Science Department Naval Postgraduate School Monterey CA 93943-5000	1
LCdr Donald P. Brutzman Code OR/Br Operations Research Department Naval Postgraduate School Monterey CA 93943-5000	1
Robert A. Peterson Image Information Program, FGM Eastman Kodak Company 1447 St Paul Blvd Rochester NY 14653-7004	1
Lt. Kevin R. Peterson 1931 Clark Road Rochester NY 14625	2